An Expert System for Automatic Software Protection


Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (31st cycle)

An Expert System for Automatic Software Protection

Leonardo Regano
* * * * * *

Supervisors
Prof. Antonio Lioy, Supervisor
Cataldo Basile, Ph.D., Co-supervisor

Doctoral Examination Committee:
Bart Coppens, Ph.D., Referee, Ghent University
Prof. Claudia Raibulet, Referee, Università degli Studi di Milano-Bicocca
Prof. Bjorn De Sutter, Ghent University
Prof. Stefano Paraboschi, Università degli Studi di Bergamo
Prof. Riccardo Sisto, Politecnico di Torino

Politecnico di Torino
17 July 2019

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

.....

Leonardo Regano
Turin, 17 July 2019

Abstract

In the Information Age, software is omnipresent in almost everyone's life. A plethora of services is offered in a digital manner, for example with e-government portals, e-banking mobile apps, and e-commerce websites. Indeed, users trust the software enabling such services with their personal information, like credit card numbers or e-banking credentials. Thus, software must be protected with care, in order to mitigate possible threats against such valuable data. Furthermore, software companies have to protect the assets in their applications, such as intellectual property of algorithms, methods preventing unauthorized application distribution, and users' personal data. Failing to do so may deeply damage software companies' finances, due to lost application sales, and reputation, if users' personal data is leaked.

However, protecting applications is a cumbersome task, reserved to few expert practitioners of this field, due to the rising complexity of applications, and the availability of numerous protection techniques. Each of the latter has strengths and weakness, and their effectiveness in safeguarding the application assets depend on numerous factors, such as the structure of protected code, the tools employed to apply such protections along with their configuration parameters, and the expected skills of a possible attacker that may be interested in breaching the application assets.

In this thesis, an expert system for automating the protection of applications is presented. To the best of the author's knowledge, it is the first application of the expert system paradigm to this challenging problem. Mimicking the decision process of a software security expert, the system, given the source code of an application that must be protected, is able to produce a binary of the application, hardened with the protection technique most suitable to defer, for the longest time possible, an attacker aiming to breach the application assets. Apart from the program source code, the system requires from the user only a list of the assets that must be protected, with each of them associated with one or more high-level security requirements (e.g., confidentiality, integrity), which must be safeguarded against possible attacks. The system has been developed during the EC-funded ASPIRE project, whose objective was to develop a set of protection techniques for Android applications, along with automated tools to deploy them on threatened code. The system is not only able to decide the generic protection techniques that must be applied to the program code, but also the specific parameters to drive such tools, thus providing a comprehensive protection solution specifically tailored for the targeted application. The system is based on the formalization of the mental decision processes and background knowledge of software security experts involved in the aforementioned project.

This thesis advances the state of the art in the field of software security with the following contributions: (1) a meta-model for software security, able to formalize all the related concepts, such as characteristics of the application and of the components of its code, attacks that can be mounted against the application, and protection techniques that can be used to mitigate them; (2) a risk assessment methodology for software, with a formalization of attacks against applications assets, consisting in the identification of simple attack tasks, which can be then chained incomplete attacks that can be carried out to successfully breach the application assets; (3) a risk mitigation strategy, based on a game-theoretic approach, able to infer the protections best able to defer possible attacks against the application; (4) a set of asset hiding strategies, devised to increase the effort needed by an attacker to locate assets in the application binary; (5) a complete and automated workflow for software protection, implementing the aforementioned risk management processes in a fully-fledged expert system.

Contents

List of Tables VI

List of Figures VII

1 Background 7
- 1.1 Software protection . . . 7
  - 1.1.1 Automated protection tools . . . 7
  - 1.1.2 Anti-reverse engineering techniques . . . 9
  - 1.1.3 Anti-tampering techniques . . . 21
  - 1.1.4 Anti-debugging . . . 26
- 1.2 Knowledge-based and expert systems . . . 27
  - 1.2.1 Knowledge bases and ontologies . . . 27
  - 1.2.2 Inference engines . . . 29
  - 1.2.3 Expert systems . . . 30

2 Decision support system for software protection 33
- 2.1 Problem statement . . . 33
  - 2.1.1 Risk framing . . . 34
  - 2.1.2 Risk assessment . . . 35
  - 2.1.3 Risk mitigation . . . 36
  - 2.1.4 Risk monitoring . . . 38
- 2.2 Automated workflow for software protection . . . 39
  - 2.2.1 Software protection meta-model . . . 39
  - 2.2.2 Risk assessment phase . . . 41
  - 2.2.3 Asset protection phase . . . 42
  - 2.2.4 Asset hiding phase . . . 43
  - 2.2.5 Complete workflow . . . 44
- 2.3 Workflow execution example . . . 45
- 2.4 Validation . . . 47
  - 2.4.1 Qualitative evaluation . . . 48
  - 2.4.2 Experimental assessment . . . 50

3 Software security meta-model 52
- 3.1 Core meta-model . . . 53
- 3.2 Application meta-model . . . 55
- 3.3 Protection meta-model . . . 56
- 3.4 Attack meta-model . . . 57
- 3.5 Meta-model validation . . . 59

4 Risk assessment 61
4.1 Application structure modeling . . . 63
4.2 Attacker goals modeling . . . 63
4.3 Attack steps and paths modeling . . . 64
4.4 Risk probability . . . 66
4.5 Validation . . . 67

5 Asset protection 69
5.1 Protection decision workflow . . . 72
5.2 Deployed protection instances enumeration . . . 73
5.3 Inference of valid solutions . . . 75
5.4 Software metrics . . . 77
5.4.1 Software protection complexity metrics . . . 78
5.4.2 Complexity metrics prediction . . . 80
5.4.3 Protection overhead estimation . . . 82
5.5 Asset protection solution inference . . . 84
5.5.1 Protection index . . . 86
5.5.2 Solution solver mini-max algorithm . . . 89
5.6 Validation . . . 91

6 Asset hiding 93
6.1 Protection fingerprints . . . 94
6.2 Asset hiding strategies . . . 95
6.3 Mixed Integer-Linear Programming model . . . 97
6.3.1 Application structure and protections . . . 97
6.3.2 Domain Parameters . . . 99
6.3.3 Linear Problem . . . 100
6.4 Translation algorithm . . . 104
6.5 Validation . . . 105

7 Conclusions and future work 106

A ESP implementation 108
A.1 Main ESP components . . . 108
A.2 ESP workflow . . . 109
A.3 Solution deployment . . . 110

Bibliography 115

List of Tables

2.1 Functions marked as assets, grouped by Sumatra phases. 46
2.2 Classes automatically instantiated in the source code analysis phase of Expert system for Software Protection (ESP) workflow, executed on the Sumatra application. 46
2.3 Protection solution inferred by ESP for the Sumatra application. 48
2.4 SLOC of ASPIRE use-case applications, used for the ESP validation. 49
2.5 Code statistics of applications used for ESP experimental assessment. 50
5.1 Software complexity metrics supported by ESP. 78
5.2 Code regions in the metrics prediction data set. 82
6.1 Asset hiding strategies for the protection techniques employed by ESP. 95

List of Figures

1.1 Control flow graph before and after the control flow flattening obfuscation. 12
1.2 Control flow graph of binary code resulting from compilation, after source-to-source flattening, of a `if...else` statement. 14
1.3 Basic block splitting with an opaque predicate: the condition is always true, therefore the branch will be always taken. 16
1.4 Basic block splitting with an opaque predicate: its evaluation is not known a priori, but both possible flows result in the same program logic. 18
2.1 ESP workflow. 45
2.2 ESP execution times on applications reported in Table 2.5, with increasing number of available Protection Instances (PIs). 51
2.3 Execution times for ESP risk assessment phase on applications reported in Table 2.5. 51
3.1 UML class diagram of the core meta-model. 54
3.2 UML class diagram of the application meta-model. 55
3.3 UML class diagram of the protection meta-model. 57
3.4 UML class diagram of the attack meta-model. 58
5.1 ESP protection decision process, with main software protection meta-model classes and ESP components involved in the process. 73
5.2 Cyclomatic Complexity (CC) and Halstead's Length (HL) predictions for "branch functions, high overhead" PI. 82
5.3 Example execution of a mini-max algorithm. 86
5.4 Example of a mini-max tree built by the solution solver. 90
6.1 Control flow graph before and after the control flow flattening obfuscation. 94
6.2 Code regions tree representing a C source code. 98

Introduction

> I guarantee that whatever (protection) scheme you come up with will take less time to break than to think of it.
>
> Philip Don Estridge

Over the last 50 years, software has evolved from being a technology obscure to most people, used to solve complex problems in military and research (e.g., guidance of the Minuteman intercontinental ballistic missiles [130] and of the Apollo 11 spacecraft [78]), to a fundamental part of almost everyone life. This was due to numerous breakthroughs in the IT field, from the successful commercialization of first personal computers [61] (e.g., Altair 8800, Apple I) to the development of the World Wide Web [20], and more recently the widespread adoption of smartphones[1]. Leveraging these advancements, in the last years an impressive amount of services have been digitized, for example trough e-government portals, e-banking apps and media streaming services. Indeed, people trust the software employed to offer these services with sensitive personal information, like credit card and routing numbers, health-related data, and, in countries adopting e-voting systems (notably, the U.S.) even their political orientation. However, this valuable private data is frequently endangered by cyber-attacks, leveraging vulnerabilities of the software used to enable these services. Governments and companies have suffered massive data leakages. Recent examples are the account data exposure of millions of Facebook users[2], and the Equifax data breach[3], resulting in millions of U.S., Canadian and U.K. consumers credit rankings exposed. Sensitive data is targeted by malicious actors either internal (e.g., disgruntled employees) or external to the organization (e.g., criminal groups, state-affiliated actors, solo attackers)[4]. Such actors leverage a variety of vulnerabilities of attacked organizations IT ecosystems, for example misconfigured network security devices (e.g., firewalls), human errors (e.g., weak passwords) and presence of malware on the targeted organization user devices (computers, but also personal mobile phones due to BYOD policies).

A common cause of the latter vulnerability is the use of unlicensed proprietary applications on such devices. The 2018 BSA Global Security Survey estimated that, worldwide, 37% of proprietary software copies are unlicensed. Attackers develop *cracks* to circumvent the license checks embedded in such applications, and then redistribute them (for example via BitTorrent sites), after having inserted malware in them. When private users or companies illegally download and

[1] https://www2.deloitte.com/content/dam/Deloitte/us/Documents/technology-media-telecommunications/us-global-mobile-consumer-survey-second-edition.pdf

[2] https://www.upguard.com/breaches/facebook-user-data-leak

[3] https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832

[4] https://enterprise.verizon.com/resources/reports/DBIR_2018_Report.pdf

use such unlicensed application, they open themselves to a plethora of attacks, for example Remote Access Trojans (RATs), opening a backdoor that gives to the attacker complete control of the infected system, or keyloggers, which register and send to the remote attacker every user keystroke (including account logins, credit cards numbers, private messages). For companies, attacks enabled by the use of pirated software has a tremendous economic impact. The aforementioned BSA report indicates a cost of 10,000 $ per infected computer, and a global cost for companies worldwide of 359 billion dollars per year. Conversely, software companies are deeply hit by lost application sales, with an estimated global unlicensed software commercial value of 46 billion dollars.

Some attackers, belonging to the so-called *warez scene*, are not driven by economic motivations; instead, they compete to be the first to crack a given application, just for fun and glory [74]. However, other malicious actors redistribute applications cracked by such "benign" attackers, inserting malware in them for financial gain [81]. Regardless of the attacker motivations, companies must protect the valuable assets in their application. Protection is needed not only to prevent the circumvention of licensing schemes, adopted to safeguard their revenue from application sales, but also to defend the company Intellectual Property (IP), i.e., algorithms, comprised in the endangered software, which enable the company to have a technical advantage against its competitor, are undoubtedly valuable. Clearly, such algorithms can be protected juridically, through the registration of patents covering the algorithms. However, if another company manages to steal such information, the ensuing litigation due to the infringed patent could last years, such in the case of the patent wars between Apple and Samsung [44]. Finally, to prevent leakage of user sensitive information, protection is also needed for data structures holding such information. If a program (e.g., an e-banking application) is executed on a compromised user device, an attacker controlling the latter remotely (e.g., with a RAT) could attack the application, purloining such sensitive information (e.g., credentials to access the bank account). Another kind of sensitive data, contained in an application running on an untrusted device, which must be protected, are the keys of cryptographic algorithms used to secure communications of the application over the Internet. For example, an instant messaging app on a mobile device can use end-to-end encryption (as in the case of WhatsApp[5]) to prevent eavesdropping of communications between two users of the application, perpetrated by a malicious actor residing on the path between the users. However, if the attacker obtains remote access of one of the users' devices, he or she can try to find the cryptographic key used to secure the communication, thus rendering encryption useless.

Indeed, the first software protection techniques were devised contemporaneously to the diffusion of personal computers, when the physical means of distribution of applications were floppy disks or audio cassettes, and users started to make and distribute copies of the original medium, instead of legally buying the applications [58]. Especially floppy disks were an enabling medium for software piracy, due to their lower cost for user w.r.t. audio cassettes [102]. These techniques tried to prevent successful copying of disks with the insertion of bad sectors, i.e., voluntarily damaged small sections of the original disk, whose presence was checked by the application at boot. The consumer disk drives were not able to reproduce these sectors on the copied disks, and, when a user started an illegitimate copy of the application, the latter would not find the bad sectors and would subsequently halt immediately after boot. However, crackers responded by merely finding and subsequently removing the code responsible for the bad sector check in the application. An even more naive technique was checking that the user physically possessed the manual shipped with the application, e.g., by asking at run-time the user to provide the first word at a given line and page of the manual; obviously, this protection could be easily circumvented by photocopying the manual, and was inconvenient for legitimate users. Through the

[5] https://faq.whatsapp.com/en/android/28030015/

years, numerous copy protection techniques were proposed, and constantly defeated by crackers. A more resistant copy protection was hardware-based, consisting of a serial port dongle, tasked by the protected application to execute some instructions. Circumventing this kind of protection was more difficult, since the hardware dongle itself was too costly to reproduce by crackers, and involved the identification of all instructions executed in the dongle, and the writing of a software emulator that executed the instructions, fooling the application in believing that the hardware dongle was connected to the user pc. A comprehensive survey of the copy protection techniques available at the time was presented by Gosler in 1985 [60].

Modern software protection was introduced after 1990, with the seminal works of Cohen [35], Goldreich and Ostrovsky [59], and Aucsmith [9]. The first two introduce the family of techniques known as software obfuscation, aiming at rendering the application code difficult to read and understand by an attacker. A comprehensive taxonomy of such techniques was presented by Collberg [37]. Aucsmith paper is the first to present the concept of tamper-resistant software, i.e., applications protected with techniques able to identify any modifications to their code, and to react at this tampering attempts, for example terminating instantly the execution, or with *graceful degradation*[6]. A fundamental assumption of such techniques is that, in the so-called man-at-the-end (MATE) scenario, where the attacker operates the application on its device, he or she has *white-box access* [33] to the application. Without any protection applied, the attacker has complete access to the program code, being able to analyze and modify it at will, and to the application execution, again with the possibility of analyzing it, for example observing the instructions executed by the processor through a debugger, and tamper the content of application memory. Indeed, in such context, a perfect protection does not exist. An attacker, if willing to spend the requested time, will be able to circumvent such protections, as shown formally by Barak *et al.* [11]. However, as noted by Cohen [35], a possible business model for software companies involves deferring of such attacks for a time sufficient to sell enough copies of the product, and possibly to release a new version of the application, protected in a different way in order to force the attack process to start again from scratch.

Nowadays, many different protection techniques are available, falling in the two aforementioned families. However, protecting software remains a cumbersome task, reserved to few experts, working in software security companies. The reasons for this are various. First, choosing the right protection techniques requires a deep knowledge of attackers methodologies and used tools (e.g., decompilers, debuggers, memory scanners), in order to understand the possible attacks against the application assets. Furthermore, the effectiveness of protections in safeguarding specific functions or data depends on the characteristics of the program code, the specific attacks that can be mounted on them, and the specific implementation of such protection techniques. Thus, the right protections for the targeted applications are still decided manually by experts.

This was the case also of computer network protection, for example for the task of configuring network access control devices, like for example firewalls. Experienced system administrator manually chose the rules for access control, and configured devices accordingly. However, abundant research showed that writing manually these rules was an error-prone task, and formal methodologies to assess the correctness of such rules were possible [14, 109]. In the same field, expert system, i.e., decision support systems aiming to solve complex problems reasoning on formalized human expert knowledge, were devised to monitor networks and identify intruders automatically[6, 18]. Surprisingly, the same approach has, to the best of the author's knowledge, never been applied to the complex problem of the protection of software applications.

This thesis aims to advance the state of the art in software security, presenting ESP, the first

[6] A reaction strategy to tampering attempts. At first, the application works normally, in order to lure the attacker into believing that the application code has been successfully modified, but over time, program functionalities stop working, leading in the end to a useless application.

application of the expert system paradigm to software protection. Starting from the source code of an application that must be protected, ESP is able to mimic the behaviour of a software security expert, assessing threats against the application assets, choosing among possible protections the ones best suitable to mitigate the aforementioned threats, and combining them in a comprehensive *protection solution*, i.e., combination of protections, able to defer for as most as possible the breach of application assets security. The system carries out these tasks automatically. The user needs only to specify the functions and variables in the code that constitute the program assets, and their security requirements (e.g., confidentiality, integrity). The system has been first devised and implemented during the EC-funded Advanced Software Protection: Integration, Research and Exploitation (ASPIRE) project. Its objective has been the definition and implementation of a set of automated protection tools, able to protect the software by modifying its source code without user intervention. The system has been based upon the knowledge of experts working on top-notch software security companies.

Using the automated protection tools designed in ASPIRE, ESP is not only able to define the best combination of protections, but can actually deploy such protections to the application, thus implementing a complete automated workflow that, starting from the target application source code, leads to a protected binary of the application, ready for distribution to users. After the end of the project, ESP has been extended to support Tigress[7], an automated obfuscator designed at the University of Arizona.

Due to this automated workflow, utilization of ESP is possible also for developers not introduced to the intricacies of software protections, thus enabling them to protect their applications. However, ESP can be also a powerful tool in the hands of software security experts, which can save precious time obtaining a first solution, of which they can assess the effectiveness in deferring a possible attacker, and that they can further refine with other protection techniques, at their discretion.

Due to the original scope of the ASPIRE project, which targeted mobile applications for the Android OS, the ASPIRE protections can be deployed only for applications written in C and designed to run on ARM processors; Tigress, instead, supports also the x86 Central Processing Unit (CPU). Since ESP has been tested thoroughly in the ASPIRE project by experts of the aforementioned software security companies, its experimental validation holds only for ARM applications; however, the formal reasoning processes, based on a generalized model of the application, behind its decision are not architecture-dependent; similarly, ESP supports only C applications for now, but can be easily extended, due to its modular design and generalized application model, by simply adding a new source code analysis engine for the desired programming language.

Finally, ESP integrates a novel software security strategy, which aims to solve the problem of *protection fingerprints*. When deployed to the code constituting the application assets, protection techniques may introduce a pattern in the code, or a peculiar behaviour during application execution, which can be easily recognized by the attacker, and leveraged to find the code containing the valuable assets in which the attacker is interested. To solve this problem, the ESP workflow includes an asset hiding phase. In addition to protecting the application assets' code, this strategy applies protections also to areas of code that are not sensitive from the security point of view. In this way, the attacker may be easily misled in believing that a heavily protected code contains an asset, but, after spending time to remove these protections, realizes that he or she has lost time on a regular code, with no value for him or her.

The contributions of this thesis to the existing state of the art in software protection are:

- the first comprehensive meta-model for software security, modelling all the concept involved

[7] http://tigress.cs.arizona.edu/

in the software protection process, comprising the application structure, the model of possible attacks, and of protection techniques, with their actual implementations enforced by automated protection tools;

- a formal risk assessment methodology for software, based on attack graphs;
- a formal risk mitigation methodology for software, based on a game-theoretic approach;
- a novel software security strategy to increase the effort needed by an attacker to locate assets in the application binary, based on the protection of code areas not sensitive from a security point of view;
- a complete automated workflow for software protection.

As a final remark of this (not so) brief introduction, the author of this thesis wants to highlight that ESP is the result of a long-lasting collaboration with Daniele Canavese, PhD, under the technical supervision of Cataldo Basile, PhD. Thus, all the merits resulting from the research detailed in this thesis are to be shared with them.

Bibliographic foundation

The complete workflow for software protection and the formal risk mitigation methodology described in this thesis are unpublished. However, the other methodologies part of the workflow have been the subject of the following publications:

- the risk assessment methodology has been reported in "Towards Automatic Risk Analysis and Mitigation of Software Applications", presented at the 2016 IFIP International Conference on Information Security Theory and Practice [104];
- the asset hiding strategy has been described in "Towards Optimally Hiding Protected Assets in Software Applications", presented at the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS) [105];
- a method to predict the complexity metrics of a code after protecting it with a given technique, without actually deploying the protection to the code, used by the risk mitigation methodology to evaluate the effectiveness of protections when applied to specific code areas, have been described in "Estimating Software Obfuscation Potency with Artificial Neural Networks", presented at the 13th International Workshop on Security and Trust Management (STM 2017) [29];
- the meta-model for software security has been presented in "A meta-model for software protections and reverse engineering attacks", published in the Journal of Systems and Software (Elsevier) [15].

Thesis organization

This thesis is structured as follows:

- Chapter 1 describes the protection techniques supported by ESP, and the paradigm of expert system, with an insight on their previous applications for cybersecurity;
- Chapter 2 contains a bird's eye view of ESP, describing its requirements, a high-level view of the software protection workflow, and a report of ESP validation by software security experts during the ASPIRE project;

- Chapter 3 details the software security meta-model, used to formalize all the data used throughout the software protection workflow;
- Chapter 4 describe the software risk assessment methodology, which enables ESP to automatically infer the possible threats against the analyzed application assets;
- Chapter 5 elaborates on the software risk mitigation methodology, which is used by ESP to decide the best combination of protection against the possible attacks endangering the application assets;
- Chapter 6 presents the problem of protection fingerprints, and a methodology to infer the additional protections, deployed on non-asset code areas, best able to hide the application assets location;
- Chapter 7 contains a set of concluding remarks, listing also possible future research in the software security field that may stem from further development of ESP;
- Appendix A reports the main information on ESP implementation.

Acknowledgments

First, I would like to thank prof. Antonio Lioy for the great opportunity of joining the TORSEC group and working all these years with such brilliant people.

I am very grateful to Aldo Basile, my co-advisor and mentor, for all his support and patience throughout the PhD. Many thanks to Daniele Canavese for these wonderful years of research together, during which the foundations of this thesis have been laid. Furthermore, my deepest thanks to both for reviewing this thesis in such a short time frame. Thanks also to Alessio Viticchié for sharing this adventure with me, and for reminding me to take it easy sometimes.

On a personal note, *thanks for everything* to my friends in Bari, and also thanks to my friends in Turin for having made me feel at home in the far north.

Last but not least, a big thank you to my parents, sister and brother-in-law, for their unconditional confidence and support in all these years of study and research.

Chapter 1

Background

> Any fool can know. The point is to understand.
>
> Albert Einstein

This section presents the state of the art in literature regarding the software protection techniques adopted by ESP, also presenting information about the supported implementations of such protections. Furthermore, this section provides a general background on expert systems, with a focus on previous works describing tools of this kind for computer and network security purposes.

1.1 Software protection

This section presents the software protections employed by ESP. Such techniques are categorized in function of the type of attack on software that they try to prevent. Section 1.1.2 presents the protections against reverse engineering of software. Their objective is to slow an attacker trying to understand the logic of protected code. Section 1.1.3 depicts the techniques able to identify, during the execution of a protected application, if an attacker has tampered with its code. Finally, Section 1.1.4 presents an anti-debugging technique, able to stop an attacker from attaching debuggers to a protected program, so that uses of such tools for reverse engineering or tampering purposes can be prevented.

1.1.1 Automated protection tools

To protect a target application, its code must be modified, for example altering its control flow graph (in the case of Control Flow Flattening (CFF), presented in the Section 1.1.2), or including other functions in the application (e.g., the appraiser code for remote attestation, a technique presented in Section 1.1.3). This must be done without altering the semantics of the application code, i.e., the correctness of the target application execution and of its results. Indeed, this is not easy avoiding this when manually protecting an application. Human errors in this process can easily happen, for example due to the ripple effect [30, 23]. Even small modifications to the code of a specific module of an application may cause the latter to assume unwanted behaviours during execution. Therefore, automating this process is certainly desirable, avoiding the insertion of bugs in the code while protecting it. A notable example of automated protection tool is the Irdeto Cloakware Transcoder [85], a proprietary obfuscator for C/C++ code.

In the remainder of the section, three automated protection tools are presented; these are leveraged by ESP, after choosing the most suitable protections to defend the application against

possible attacks, to deploy such protections to the target application code. Many of the protections techniques supported by ESP have been developed during the ASPIRE project.

One of the most powerful tools used by ESP is DIABLO (Diablo Is A Better Link-time Optimizer) [122], a link-time rewriting framework, able to analyze and transform the binary objects constituting an application, when such objects are being linked. Information produced by the linker (e.g., how relative addresses of instructions and data of relocatable object files are translated into absolute addresses in the final linked executable) is leveraged by Diablo Is A Better Link-time Optimizer (DIABLO) to build an abstract representation of the program code, independent from the Central Processing Unit (CPU) architecture for which the code has been written (e.g., ARM, Intel x86). However, it also includes optimizations for specific CPU architectures. In this way, transformations and analyses on binaries can be devised on this representation, without taking into account the subtleties of a specific instruction set. DIABLO has been employed for a variety of purposes, such as program compaction, reducing power consumption and code instrumentation. In particular, ESP employs the binary obfuscation transformations built on top of DIABLO during the ASPIRE project. These will be detailed in the following sections. Given the ASPIRE project scope, DIABLO obfuscations have been devised for Android applications running on ARM CPUs. Also, optimizations for the ARM architectures are more advanced w.r.t. the ones implemented for Intel x86 CPUs. Thus, ESP will use DIABLO transformations only when protecting such programs.

Deployment of this protection techniques is automated using the ASPIRE Compiler Tool Chain (ACTC), a Python 3 script that, given the code location that must be protected, and the technique that must be used to do so, produces a protected binary from the target application source code. Examples of tasks automated by the ACTC are the invocation of DIABLO with the appropriate parameters to binary-level obfuscation techniques (see Section 1.1.2), or the inclusion at compile-time of libraries needed by the deployed protections. When protecting a program, the ACTC follows these three main steps (a detailed description of the ACTC workflow is available in [12]):

1. applies source-level protections, editing the target application source code, e.g., to insert the appraiser in the target application when applying remote attestation (see Section 1.1.3);

2. compiles the source code; since target applications are written for devices equipped with ARM CPUs, the ACTC supports cross-compilation[1] in order to compile ARM applications on Intel machines, using for example the cross-compilation toolchains provided with the Android Native Development Kit[2];

3. applies the binary-level protections, rewriting the binary using DIABLO, e.g., to apply the code obfuscation techniques described in Section 1.1.2.

To instruct the ACTC on the areas of code that must be protected, and on the protections that must be used, the target application developer must mark such code and variables with a set of code annotations, specified with the `pragma` C directive[3], which are parsed by the ACTC at the start of its workflow. A detailed description of the annotations supported by the ACTC is reported in [12]; taking the password check example from Section 1.1.2, the annotation in Listing 1.1 tells the ACTC to apply the binary CFF technique with DIABLO on all the code comprised between the `pragma` directives.

[1] Cross-compilation is a technique that permits to compile an application on a machine running a different Operating System (OS) or CPU architecture w.r.t. the one on which the obtained binary will be executed.

[2] https://developer.android.com/ndk/guides/standalone_toolchain

[3] https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html

```c
#include <stdio.h>
#include <string.h>

char pwd[] = "hardcodedPassword";

int main()
{
        char temp[20] = "";

        _Pragma("ASPIRE begin protection
                (obfuscations,enable_obfuscation(flattening))");
        printf("Insert password: ");

        scanf("%20s",temp);

        if(strcmp(temp,pwd)==0)
                printf("Correct password!\n");
        else
                printf("Wrong password!\n");
        _Pragma("ASPIRE end");

        return 0;
}
```

Listing 1.1: C code marked with ACTC annotation to flatten comprised code.

Apart from the techniques developed during the ASPIRE project, ESP supports also the code and data obfuscation techniques deployed by Tigress, an automatic obfuscator for C applications. Instead of applying transformations on the binary code (such as DIABLO), Tigress operates on the application source code. In particular, the latter is preliminary transformed in C Intermediate Language (CIL) [95], a subset of the C language that is characterized by a reduced number of syntactic forms, thus being more easily manageable in an automated fashion. For example, all the different loop syntaxes supported by standard C (for, while, do) are transformed into a specific `while(1)` construct, with the loop termination handled by an explicit break statement. Then, Tigress performs the transformations requested by the user on the simplified CIL code, producing the obfuscated source code. As claimed by its authors, Tigress design is similar to the aforementioned Irdeto Cloakware Transcoder [85]. Tigress can obfuscate applications written for Linux, either for ARM or x86-64 CPU architectures.

1.1.2 Anti-reverse engineering techniques

Reverse engineering can be defined as "the process of extracting the knowledge or design blue-prints from anything man-made" [51]. Indeed, it has been a common practice in a variety of fields, such as mechanical engineering [26], biology [41] and even for military purposes [120]. Similarly, software reverse engineering, also known as program comprehension or understanding, is "the process of identifying software components, their inter-relationships, and representing these entities at a higher level of abstraction" [96]. Software reverse engineering can be legitimate. An example is a software developer that has to interface its code with an open source Application Programming Interface (API) that is poorly documented, and therefore has to understand the API 's code to properly use it.

From a legal point of view, reverse engineering is legitimate, unless, in the case of a proprietary application, it is explicitly prohibited by the reversed software EULA[4]. However, European Union [40] and U.S.A. [1] laws regulating the rights of software providers explicitly allow reverse engineering of proprietary software, if it is carried out for interoperability purposes. A well-known example of this caveat is the Samba[5] free software, which enables Linux machines to offer or use network file sharing and printing services based on the Microsoft Server Message Block (SMB)[6] (Server Message Block) proprietary network protocol: Samba development has been carried out essentially via reverse engineering of the SMB protocol[7].

Still, software companies need to counter reverse engineering of their applications, in order to preserve the assets that, as said in Section 1.1, constitute the business value of the software. Indeed, distributing only the application binaries to end-users is not sufficient, since, as detailed in Chapter 4, attackers may easily obtain the assembly code from a binary, by means of a disassembler, or even a reconstruction of the application's original source code, using a decompiler. For example, the well-known commercial debugger Hex-Rays IDA[8] supports both disassembling and decompilation (via a separate plug-in[9]).

Various protection techniques have been devised in order to increase the difficulty of the reverse engineering process. They fall under the umbrella term of software obfuscation [37]. Essentially, the objective of such techniques is to transform the binary (or the sensitive parts of the binary) that must be protected into an obfuscated version that, whilst behaving in the exact same way as the original binary when executed, is difficult to understand by a human being. Similarly, obfuscation can be applied to constants or variables in the code, in order to mask their real value when the application is inspected either statically (especially for constants), for example looking at the application reconstructed assembly or source code, or dynamically, inspecting the execution by means of a debugger.

It is theoretically possible to apply obfuscation by hand, and even to directly write obfuscated code. However, this is typically a recreational activity. There are even competitions, like the International Obfuscated C Code Contest (IOCCC)[10], where programmers compete in writing the best obfuscated code. In the software industry, where applications must be protected against reverse engineering, it is more common to rely on automated obfuscation tools, e.g., the aforementioned Irdeto Cloakware Transcoder [85]. The remainder of this section details the obfuscation techniques supported by ESP, provided by the automated protection tools detailed in Section 1.1.1. Possible attacker behaviours are used in the explanation of the presented techniques for the sake of clarity. While a complete model of such behaviours is still not present in literature, an initial work towards this has been made by Ceccato *et al.* [31].

Code obfuscation

This section describes the code obfuscation techniques supported by ESP. The main objective of such techniques is to harden the comprehension by an attacker of the protected code. All the

[4]End User License Agreement: the contract between the software provider and the end-user, indicating the rights obtained by the latter after purchasing from the first a license for the software.

[5]https://www.samba.org/

[6]https://docs.microsoft.com/en-us/windows/desktop/fileio/microsoft-smb-protocol-and-cifs-protocol-overview

[7]https://www.samba.org/samba/docs/SambaIntro.html

[8]https://www.hex-rays.com/products/ida/index.shtml

[9]https://www.hex-rays.com/products/decompiler/index.shtml

[10]https://www.ioccc.org/

following transformations are deployed by ESP via DIABLO[11] or Tigress[12], with some techniques supported by both.

CFF is a transformation that, when applied to a function or, generically, to a code area, hides its original control flow, modifying its structure in a form that increases the effort needed by an attacker to understand the protected function logic. Indeed, analyzing the control flow is one of the basic activities a reverse engineer must undertake, to understand an algorithm. The control flow is usually obtained in an automated fashion, and is one of the most typical features of static program analysis tools, able to analyze an application code without actually executing it; control flow can be reconstructed both from source and binary code. The Control Flow Graph (CFG) is the most common human-readable representation of such flow; its nodes, called *basic blocks*, contain sequences of consecutive instructions, so that if the first instruction of a basic block is executed, the other instructions in the basic block must be executed as well, in the specified order. Therefore, a basic block code must have:

- exactly one entry point, so that no instruction after the first one in the block must be a target for any jump instruction in the whole application;
- exactly one exit point, the last instruction of the block, which is the only one that can cause the program execution to jump to another basic block.

The basic algorithm to "flatten" a function, first described by Wang *et al.* [126] and subsequently formalized by László and Kiss [82], is the following:

1. the function body is split into basic blocks;
2. a selective structure (e.g., the `switch` statement in the C language) is set-up, with a number of `case` statement equal to the number of basic blocks obtained in the previous step;
3. the selective structure is wrapped in a loop statement (e.g., `while` in C);
4. each basic block code is put into a different `case` statement;
5. a control variable, which is checked by both the `while` and `switch` statements, is responsible to ensure the original control flow of the protected code; it is set at the end of each basic block, i.e., `case` statement, in order to select the next basic block that must be executed (via the `switch` statement), and finally to terminate the protected code execution (by negating the condition of the `while` statement).

Flattening may be deployed by ESP either on a source level, using Tigress, or on a binary one, thanks to DIABLO. For an example of flattening (source-to-source for simplicity), consider the code in Listing 1.2, a (rather naive) C implementation of a password check. Flattening the `main` function would lead a result like the one in Listing 1.3.

Excluding the initial variable declarations, the `main` body contains five basic blocks, which appear in each `case` statement of the flattened version[13]. The password insertion by the user with its comparison with the hardcoded password, the branch depending on the comparison result, and the two possible outcomes in the cases of a correct or a wrong password inserted by the user. The

[11] https://aspire-fp7.eu/sites/default/files/D2.06-Binary-Code-Obfuscation-Report.pdf

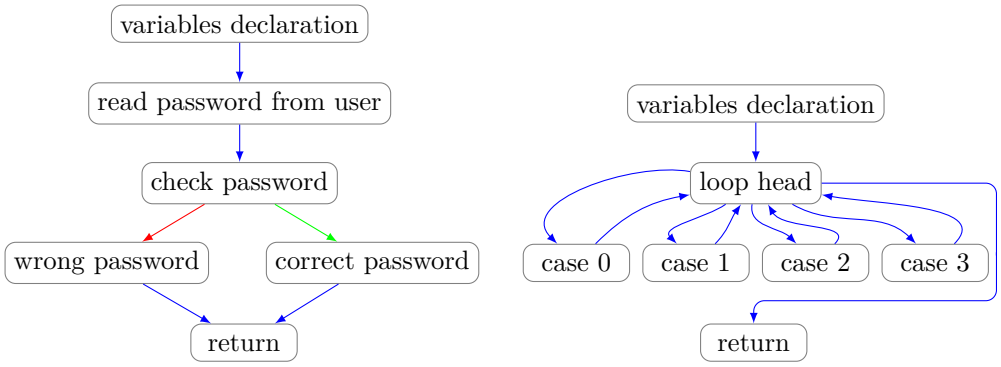
[12] http://tigress.cs.arizona.edu/transformPage/index.html

[13] Note that the `case` arguments in the example follow the actual control flow for the sake of simplicity; real implementations use arbitrary numbers for the control variable, and, for source-to-source transformations, randomize the order of `case` compound statements.

```c
#include <stdio.h>
#include <string.h>

char pwd[] = "hardcodedPassword";

int main()
{
        char temp[20] = "";
        printf("Insert password: ");
        scanf("%20s",temp);

        if(strcmp(temp,pwd)==0)
                printf("Correct password!\n");
        else
                printf("Wrong password!\n");
        return 0;
}
```

Listing 1.2: C code for a password check.

`control` variable drives the flow of the program. In the `case 1` body is responsible for displaying the correct message depending on the result of the password comparison, and also ensures that, either reaction is taken, the program terminates, thanks to the condition in the `while` statement. Figure 1.1 shows the effect of CFF obfuscation on the CFG of the protected code. While on the original graph the consequentiality of the basic block is evident (read the password from the user, check it, and print the appropriate message), understanding the flow of the flattened version of the code is more difficult, since all the basic blocks are parallel in the graph. Therefore, to understand the flow of a flattened function, an attacker cannot just rely on the CFG structure, but must actually analyze the code to reconstruct the original function structure.



(a) original (b) flattened

Figure 1.1: Control flow graph before and after the control flow flattening obfuscation.

It should be noted that, for source-to-source flattening, the actual algorithm differs slightly from the one stated before. Looking at the `case 1` of the obfuscated example from before, the binary equivalent of the case after compilation may comprise[14] three basic blocks, as shown in

[14]The exact result of the compilation depends on various factors, including the compiler used and the chosen level of optimization.

```c
#include <stdio.h>
#include <string.h>

char pwd[] = "myPassword";

int main()
{
        char temp[20] = "";
        int strcmp_result = 0;
        int control = 0;

        while (control != 4) {
                switch (control) {
                case 0:
                        printf("Insert password: ");
                        scanf("%20s", temp);
                        strcmp_result = strcmp(temp,pwd);
                        control = 1;
                        break;
                case 1:
                        if (strcmp_result == 0)
                                control = 2;
                        else
                                control = 3;
                        break;
                case 2:
                        printf("Correct password!\n");
                        control = 4;
                        break;
                case 3:
                        printf("Wrong password!\n");
                        control = 4;
                        break;
        }

        return 0;
}
```

Listing 1.3: Flattened C code for a password check.

Figure 1.2. The check on the `strcmp` result, and the two `control` variable assignment depending on the check result. Indeed, this is necessary on a source-to-source level to preserve the correctness of the resulting C code, since the `if` statement is usually followed by at least an instruction. Therefore, the flattening is applied to such statements by moving all the instructions contained in the `if` compound statement into a new `case`, and by setting the `control` variable in the `if` to preserve the original control flow, i.e., to execute the compound statement after the `if`, should the related condition be satisfied.

In [82], László and Kiss formally describe how to flatten all the C/C++ control flow statements (e.g `if`, `for`, `switch`), while preserving the resulting source code correctness.

Branch functions have been first introduced by Linn and Debray [86], as a way to decrease the accuracy of disassemblers in translating binary code in human-readable assembly. Before describing this transformation, a brief overview of the main disassembly techniques is necessary.

Disassemblers can operate in a static way, analyzing the machine code, or in a dynamic way, inspecting the execution of the binary using a debugger. The problem of the dynamic approach is that only the actually executed instructions are translated into assembly, therefore the disassembly result is dependent on the particular input given to the program. Also, dynamic

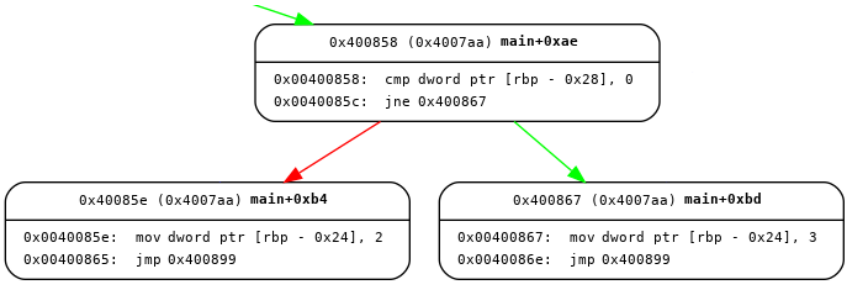


Figure 1.2: Control flow graph of binary code resulting from compilation, after source-to-source flattening, of a `if...else` statement.

disassembly is considerably slower than the static one, since it involves the actual execution of the analyzed program. Since dynamic disassembly is achieved by debugging the application, in order to trace the executed instruction, preventing application debugging automatically hinders dynamic disassembly techniques. DIABLO anti-debugging technique is described in Section 1.1.4.

Instead, branch functions are effective against static disassembly, in particular against two of the most common static disassembly techniques, linear sweep and recursive traversal. The first one, implemented for example by the GNU utility objdump[15], consists in decoding instructions linearly, starting from the first byte of the text section[16]; the address of the next instruction that must be decoded is evaluated by incrementing the actual instruction address with its expected length[17]. However, sometimes static data (e.g., constants) may be inserted into the code section [127], e.g., for performance purposes. Therefore, linear sweep disassemblers may confuse static data for instructions. On variable instruction length architectures as x86, this may lead to an incorrect translation of all the following machine code, since the next instruction address evaluation will be based on the length of the misinterpreted instruction opcode. The disassembler can become aware of its error only if the miscalculated next instruction address will point to an opcode that is invalid for the analyzed program instruction set; this may happen after an indefinite, possibly large number of erroneously decoded instructions.

To solve this problem, the recursive traversal disassembly algorithm has been devised. Instead of following the order in which instructions appear in the executable, the disassembler, when encountering a jump, will decode as next instruction its target; in general, the disassembly order will follow the program control flow. In case of branches, all the possible targets, and consequently the possible resulting flows, are followed. Indeed, this tackles the issues related to the presence of constants in the text section, since these will never be the target of jumps. However, this approach presents another problem, the handling of indirect jumps. The latter are the result of jump instructions with a target that is not explicitly known until run-time, e.g., a `JMP EAX` x86 instruction, which will jump to the address specified in the general purpose register `EAX` when the `JMP` instruction is executing. Therefore, the disassembler would be unable to follow indirect jumps, resulting in a partial translation of the machine code. Therefore, disassemblers rely on

[15] https://www.gnu.org/software/binutils/

[16] In x86 and ARM architectures, the *text section* is the executable area containing instructions; constants and variables are stored in the *data section*.

[17] Traditional ARM instructions have a fixed length of 32 bits, while the Thumb and Thumb-2 modes feature respectively 16 bits and a mix of 16 and 32 bits instruction widths; x86 instructions have a variable length, depending on the number of operands expected by the specific instruction (e.g., 2 operands for the `MOV` instruction).

extensions of the basic recursive traversal algorithm to handle the most common cases when compilers produce indirect jumps. For example, `switch` statements of C programs are typically translated in binary via jump tables. These contain the target addresses for the first instruction of each `case`. The `switch` is translated as an indirect jump having as target one of the addresses contained in the jump tables, in function of the `switch` control variable value during the execution; in practice, this will result in a `JMP jump_addr+offset` instruction, where `jump_addr` is the address of the first element of the jump table, incremented by an `offset` equal to the size of one jump table element times the control variable value. Consequently, a disassembler may be extended to recognize this evaluation, in order to find statically the jump table in the binary, thus being able to follow all the possible targets of an indirect jump resulting from the translation of a `switch` statement.

Branch functions aim to thwart static disassembly algorithms, by transforming direct jumps into indirect ones, and by also increasing the difficulty to reconstruct the targets of indirect jumps already present in the code that must be protected against reverse engineering. This is obtained by substituting these jumps with calls to a branch function, which will be responsible of transferring the program control to the substituted jump target, preserving the original program control flow. Therefore, a branch function needs a way to select the target to which it will ultimately jump, given the code location from which it has been called. The most simple implementation may be a jump table, with the index of the correct target passed with the call to the branch function. While this could be effective in hindering automatic disassembly, it can be easily reverse engineered by an attacker, which could then develop an extension to the disassembler to handle automatically with such branch functions. Therefore, the target evaluation must be implemented in an obfuscated way. Linn and Debray [86] propose an implementation based on perfect hashing [57]. Essentially, instead of explicitly specifying in the binary code the jump table index, the latter is evaluated at run-time using a perfect hash function, which receives in input the address of the call to the branch function (e.g., the substituted jump address). In this way, an attacker should reverse engineer the perfect hash function, a task that is deemed not trivial by the authors.

Both Tigress and DIABLO support this technique. However, neither of them uses the perfect hash table implementation. In particular, Tigress authors justify their decision[18] due to the infeasibility of a source-to-source version of such technique implementation. Instead, both obfuscators implement indirect jumps by passing to the branch function the offset from the function call address to the original target address. Tigress implements this by using C pointer arithmetic, starting from the addresses of locally declared labels[19] specifically inserted for this purpose, while DIABLO store the offsets in global variables by the branch function, loaded into a random register that is subsequently read by the actual jump instruction.

Opaque predicates, introduced by Collberg *et al.* in [37], are boolean expressions that have always the same outcome at run-time (always true or always false). However, this outcome is difficult to be formally evaluated by an automatic deobfuscator[20] in a static way[21]. Therefore, if employed as the condition of a branch, is difficult to automatically derive from the predicate that always one of the branches will be taken by the program. This can be leveraged to add fake code, which will be positioned as the target of the branch that is never taken. While the actual

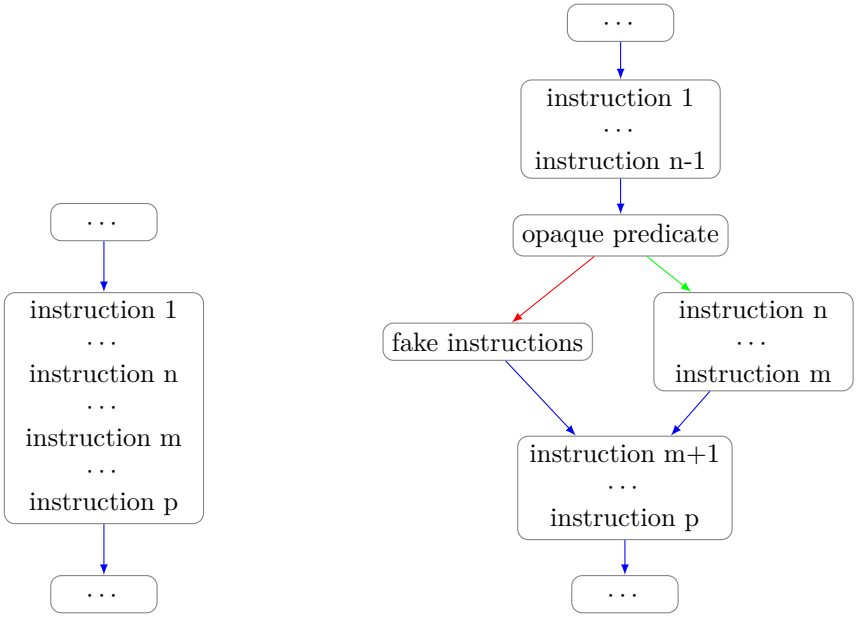
[18] http://tigress.cs.arizona.edu/transformPage/docs/encodeBranches/index.html

[19] https://gcc.gnu.org/onlinedocs/gcc/Local-Labels.html

[20] An automated tool, able to analyze obfuscated source codes or binaries in order to partially revert known obfuscation schemes, thus producing an approximation of the original sources or binaries prior to obfuscation.

[21] It should be noted that, while using a fuzzer can mark a branch as probably never taken, there is no dynamic way to be formally assured of it, since predicates are typically evaluated on unbounded variables, resulting on infinite possible inputs that should be tested by the fuzzer.

program control flow remains untouched, the amount of code that an attacker will need to analyze to understand the application will be increased, thus slowing the reverse engineering process. Also, this transformation can be exploited to split basic blocks, again to hinder comprehension of the contained code; adding an opaque predicate in the middle of the block means adding a branch, therefore resulting in four basic blocks, as shown in Figure 1.3.



(a) original (b) with opaque predicate

Figure 1.3: Basic block splitting with an opaque predicate: the condition is always true, therefore the branch will be always taken.

Again, both DIABLO and Tigress support this technique. However, the implementations differ. DIABLO relies on opaque predicates based on mathematical expressions, while Tigress uses pointer algebra.

Opaque predicates implemented by DIABLO are based on mathematical properties of the conditional expression, e.g., $x^2 \geq 0$ which is always true, since the square of a real number is never negative. While they can be evaluated rapidly at run-time, without slowing the application execution, they are designed to prevent disassemblers from formally proving them, unless the latter are instructed to recognize such hardcoded predicates by an attacker that has found them via manual inspection of the code. Therefore, while the predicates inserted by DIABLO are always comprised in the aforementioned set, the generation of the instructions implementing this predicates is randomized in two ways. Registers employed by such instructions are randomly chosen among dead registers[22] if they are available. Also, if the employed predicate includes an integer constant, it will be randomized, e.g., in the fourth predicate, the 0 can be substituted with any negative number.

Tigress instead relies on a technique, described by Collberg *et al.* in [38], which consists in evaluating conditions on pointers to data structures, with the latter specifically added to the code

[22]Registers are defined dead if, at a specific point of a function execution, they contain a value that will be not read by any instruction until function termination.

for this purpose. For example, given two completely separated linked lists `L1` and `L2`, we can define three pointers: `p1` and `p2` pointing to nodes of `L1`, and `p3` pointing to an element of `L2`. In the program code, a series of instructions that will move the pointers to other nodes, but that will make them always point to the same list, can be added in order to increase deobfuscation hardness. Therefore, we can list three opaque predicates:

1. `p1 != p3` and `p2 != p3`: this will always be true, since `p3` points to nodes in a different list of the ones pointed by `p1` and `p2`;
2. `p1 == p2`: this can be true or false depending on the initial nodes pointed by the pointer, and on the movement instructions operating on such pointers before the comparison.

The first kind of opaque predicates can be leveraged as explained before, i.e., to insert fake instructions in the never taken branch. This is implemented in various ways by Tigress. Calls to a random existing function or to ones that do not exist, the original code with random bugs inserted, or even random bytes. Instead, the second kind of predicates must be employed in a different way, since the branch taken is not known a priori, and can possibly depend on the program inputs[23]. The idea is to put the same code in both branches, but in different forms. In the example depicted in Figure 1.4, if the branch is taken a flattened version of the original code is executed, otherwise the original code is executed; therefore, taking the branch or not has no effect on the executed program logic. In this way, the amount of code that an attacker must analyze to comprehend the protected code doubles, since he or she has no way to know that the two branches actually execute equivalent code. Also, the attacker will probably spend time in understanding the branch logic, having to track all the possible modifications to the pointers (and possibly to the pointed structures), before realizing that these branches are taken randomly, without altering the original program logic.

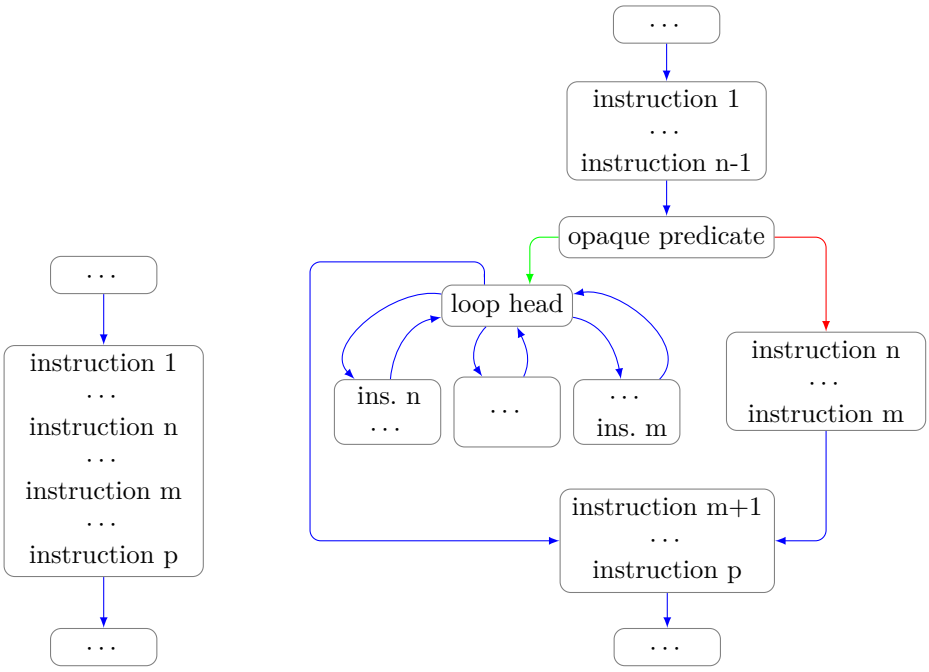
Tigress applies opaque predicates to code in a semi-automatic fashion. First, the user is responsible for selecting a function that will initialize the global data structures leveraged by predicates throughout the code. Clearly, this function must be executed before any other one containing opaque predicates; while an easy choice would be to initialize these structures in the `main` function, this would advantage the attacker, since `main` is usually the first examined function, and is therefore discouraged. Also, the user is responsible to select which functions will update pointers and data structures, and how many updates will be made by each function. Intuitively, a large number of updates will make the protection more effective, but they can slow down the application execution, especially if a function containing updates is called frequently.

Virtualization obfuscation [54] is a transformation that protects instructions by hiding their real opcodes. It is similar to executing code in a virtual machine, hence the name, but, instead of emulating an existing CPU to support program written for its architecture, translates instructions in a specially devised instruction set that, while in general preserves the original structure of instructions, uses different opcodes. At run-time, the code execution is delegated to an interpreter, which translates each instruction that must be executed from the "virtual" instruction set to the original one, so that it can be actually understood and executed by the CPU. Consequently, an attacker that wants to comprehend code protected with such obfuscation needs to reconstruct the mapping between the virtual and the original instruction set.

This transformation is deployed by ESP using Tigress. The latter, as for other transformations, implements such technique[24] in a source-to-source fashion. Given a C function that must be virtualized, Tigress first analyzes the function code to build an abstract syntax tree (AST) and the

[23] Instructions that move node pointers can be executed conditionally, depending on program input.

[24] `http://tigress.cs.arizona.edu/transformPage/docs/virtualize/index.html`



(a) original (b) with opaque predicate and flattened control flow

Figure 1.4: Basic block splitting with an opaque predicate: its evaluation is not known a priori, but both possible flows result in the same program logic.

CFG, which in turn are used to generate the function bytecode, written in the virtual instruction set, which will be interpreted at run-time. The generated bytecode is saved in the C code as raw data, using an `unsigned char` array. Then, the interpreter code is added to the program, and, throughout the code, calls to the translated functions are substituted with calls to the interpreter. To select the specific virtualized function that must be executed, a pointer to the array containing the function bytecode is passed with the call to the interpreter.

The virtual instruction set is generated at the start of the transformation randomly. This is important because, if the same instruction set would be used by Tigress for any program, an attacker that, analyzing a protected application, has been able to reconstruct the mapping between original and virtualized instruction sets, he or she would be able to construct a de-virtualizer able to automatically remove such obfuscation from any program protected by Tigress. To prevent this, a number of solutions are adopted. First, opcodes of the virtualized instructions can be randomized. Also, the instruction set may contain duplicate instructions, with different opcodes, but with the same semantic value. These instructions can be used interchangeably to translate the original C code, while this will increase the size of the resulting virtualized instruction set, thus augmenting the time needed by an attacker to reconstruct the instruction mapping. Finally, sequences of instructions can be mapped into a single superoperator [103], a virtualized instruction that, when interpreted, will result in the execution of the original instruction sequence; this solution is particularly effective in slowing instruction remapping by attackers [106].

The interpreter, to actually perform the translation at run-time, has a virtual program counter (VPC), which points to the next instruction that must be translated, and a dispatch unit, which, given the bytecode pointed by the VPC, reads the instruction virtual opcode, and calls the appropriate instruction handler. This is a function, part of the interpreter, which will actually translate the instruction in the original instruction set and will therefore execute it. An attacker

may be interested in understanding the interpreter logic, since this could help with the instruction remapping. Especially finding the instruction handler can be useful for this purpose. Therefore, Tigress protects the interpreter itself with other obfuscation techniques. For example, the dispatch unit control flow may be split by adding opaque predicates to its code.

Data obfuscation

This section presents the data obfuscation techniques supported by ESP are reported. As said in the previous sections, the purpose of data obfuscation is to prevent an attacker to understand the value of constants present in source code, when he or she analyzes the code statically, and the value of variables during the execution, if the code is inspected dynamically e.g., by means of a debugger. ESP supports two techniques to obfuscate respectively constants and variables. Both are implemented by Tigress, and therefore are source-to-source transformations.

Literals obfuscation is implemented by Tigress with two different schemas, one to obfuscate hard-coded integers, and the other applied to string literals.

To obfuscate integer constants, Tigress relies on opaque predicates, implemented with the form depicted in Section 1.1.2. For example, given two pointers `p1` and `p2` that, throughout the code, point always to two different C structures, e.g., two linked lists, the boolean expression `p1==p2` will always be false. Since the boolean false value is treated in C as a `0`, this integer value can be obfuscated easily using the aforementioned opaque predicate.

Instead, to obfuscate string literals, Tigress transforms them into calls to an "encoder" function, which will generate them at run-time. therefore, this blocks the attacker from statically searching the binary for intelligible strings. As an example, looking at the password checker from Section 1.1.2, the attacker can easily find the hard-coded password, e.g., by using the *strings* Linux command on the binary. Also, the attacker may try to elude the comparison between inserted and hard-coded password, by finding (and modifying) the `strcmp` call. Since, trying the code with a casual password, he will receive in output the *"Wrong password!"* message, he can statically analyze the code to find the address of this message in the data section, identify the specific `printf` call with that address passed as a parameter, and starting from this call location analyze the code nearby to ultimately find the `strcmp` call. By generating the strings at run-time, Tigress can prevent such (rather trivial) attacks. If the *"Wrong password!"* message is obfuscated, the string encoder function in Listing 1.4 is added to the code.

The encoder function can be used to obfuscate various string literals in the code. A specific string can be requested via the `n` integer, which will drive the `switch` statement in the encoder. The generated string is saved in the `str` parameter, a `char` array passed by reference. Obviously, the encoder function can be easily understood by the attacker. Therefore, it is strongly suggested to protect this function with the code obfuscation techniques described in Section 1.1.2.

Variable obfuscation aims to change the representation of a protected variable in memory, by means of an encoding mathematical function. When the variable is used in the code, it is decoded on-the-fly, so that its real value is present in memory for the shortest time possible. Tigress uses by the default[25] one of the encodings described in [132][75], which substitutes an integer variable v with the following:

$$v' = a \cdot v + b \tag{1.1}$$

where a and b are two random integers. When the original value must be used, it can be easily retrieved with the dual expression of 1.1:

$$v = \frac{v' - b}{a} \tag{1.2}$$

[25]Other encodings are supported, but they are trivial and therefore their use is discouraged. For example, a variable can be saved in memory xor-ed with a constant, and decoded by xor-ing it again with the same constant.

```
1  void _1_stringEncoder(int n , char str[] )
2  {
3          int encodeStrings_i3 = 0;
4          switch (n)
5          {
6                  case 0:
7                  str[encodeStrings_i3] = 'W';
8                  encodeStrings_i3 ++;
9                  str[encodeStrings_i3] = 'r';
10                 encodeStrings_i3 ++;
11                 str[encodeStrings_i3] = 'o';
12                 encodeStrings_i3 ++;
13                 str[encodeStrings_i3] = 'n';
14                 encodeStrings_i3 ++;
15                 str[encodeStrings_i3] = 'g';
16                 encodeStrings_i3 ++;
17                 str[encodeStrings_i3] = ' ';
18                 encodeStrings_i3 ++;
19                 str[encodeStrings_i3] = 'p';
20                 encodeStrings_i3 ++;
21                 str[encodeStrings_i3] = 'a';
22                 encodeStrings_i3 ++;
23                 str[encodeStrings_i3] = 's';
24                 encodeStrings_i3 ++;
25                 str[encodeStrings_i3] = 's';
26                 encodeStrings_i3 ++;
27                 str[encodeStrings_i3] = 'w';
28                 encodeStrings_i3 ++;
29                 str[encodeStrings_i3] = 'o';
30                 encodeStrings_i3 ++;
31                 str[encodeStrings_i3] = 'r';
32                 encodeStrings_i3 ++;
33                 str[encodeStrings_i3] = 'd';
34                 encodeStrings_i3 ++;
35                 str[encodeStrings_i3] = '!';
36                 encodeStrings_i3 ++;
37                 str[encodeStrings_i3] = '\n';
38                 encodeStrings_i3 ++;
39                 str[encodeStrings_i3] = '\000';
40                 encodeStrings_i3 ++;
41                 break;
42         }
43 }
```

Listing 1.4: Example of string encoder function used by the literals obfuscation technique.

While the encoding itself seems trivial, it can become interesting when applied to several variables that are subsequently combined in the code. For example, suppose we have this expression:

$$z = x \cdot y \tag{1.3}$$

and we encode all the three variables, so that:

$$x' = a_x \cdot x + b_x \tag{1.4}$$
$$y' = a_y \cdot y + b_y \tag{1.5}$$
$$z' = a_z \cdot z + b_z \tag{1.6}$$

To encode z, we should first decode x' and y' using Equation 1.2, then multiply them with Equation 1.3, and finally encode the result with Equation 1.6. However, if the decoding formulas

are explicitly used in the code to obtain x and y, an attacker could easily find such values debugging the code. However, using some modular arithmetic properties, z could be encoded by using directly x' and y' values. Using the decode formula in Equation 1.2, Equation 1.6 becomes:

$$z' = a_z \cdot \left(\frac{x' - b_x}{a_x} \cdot \frac{y' - b_y}{a_y} \right) + b_z \tag{1.7}$$

If a modulo n algebra is used, the additive and multiplicative inverse can be evaluated, so that Equation 1.2 becomes:

$$v = (v' + b^{-1}) \cdot a^{-1} \mod n \tag{1.8}$$

$$(a \cdot a^{-1}) \mod n = 0 \tag{1.9}$$

$$(b + b^{-1}) \mod n = 0 \tag{1.10}$$

Therefore, Equation 1.7 can be rewritten as:

$$z' = a_z \cdot \left((x' + b_x^{-1}) \cdot a_x^{-1} + (y' + b_y^{-1}) \cdot a_y^{-1} \right) \mod n + b_z \tag{1.11}$$

which can be simplified into:

$$z' = c \left(x'y' + {b_y}^{-1} x' + {b_x}^{-1} y' \right) \mod n + d \tag{1.12}$$

$$c = {a_x}^{-1} {a_y}^{-1} a_z \mod n \tag{1.13}$$

$$d = ({b_x}^{-1} {b_y}^{-1} c) \mod n + b_z \tag{1.14}$$

Where c and d are constants, since their components can be evaluated with Equation 1.9 and Equation 1.10, which also holds for ${b_x}^{-1}$ and ${b_y}^{-1}$; therefore, thanks to Equation 1.12 we can evaluate the encoded z' using directly the encoded x' and y', without exposing the value of x, y and z during the program execution. Similar formulas are proposed in the patent from Kandanchatha and Zhou *et al.* [75] to evaluate directly the basic boolean operations on encoded variables, without decoding them first.

1.1.3 Anti-tampering techniques

The Oxford English Dictionary defines tamper as *"interfere with (something) in order to cause damage or make unauthorized alterations."*[26]. Indeed, this definition applies well also to software tampering, especially when targeted to proprietary applications. A good example is software piracy. Attackers may be interested in obtaining an application that, during its execution, does not behave as originally intended by its authors. Motivations for this may be various. For example, given a proprietary software that should work only if a valid license is present on the user's computer, an attacker may want to alter, in the target application, the logic of the code responsible to check the license and stop the application execution if the latter is not valid. This is colloquially known as writing a *crack* for the application. Attackers may then release such modified code on Internet, either freely or for a fee (lower than the one asked by the software publishers); in the end, legitimate proprietors of the tampered software will suffer a monetary loss, due to lost sales of their application. However, attackers do not tamper software only for piracy purposes, and may not only target proprietary software. For example, an attacker may tamper with a Free and Open Source Software (FOSS) application code in order to hide a RAT[27]

[26] https://en.oxforddictionaries.com/definition/tamper

[27] https://www.owasp.org/index.php/Trojan_Horse

in it, and then redistribute the tampered program on the Internet; in this way, the attacker will obtain complete access to the machines of any unaware users that will install the free application, which can then be used at his or her pleasure, e.g. to mount Distributed Denial of Service (DDoS) attacks [90].

Thus, adding tamper resistance to software is paramount. The idea is to obtain an application that, when executed, automatically checks its code and, if notices any change to it, stops its execution. It should be noted that, similarly to reverse engineering, also software tampering can be done in a static or dynamic fashion. In the first case, the attacker modifies code prior to its execution, editing the application binary files, while in the second, the attacker first launches the execution, and then modify the application code in memory (e.g., attaching a debugger to the program), thus altering the behaviour of the program for the remainder of its execution. Therefore, checks to the application code cannot be done only when the program is started, but must span all the program execution.

ESP uses three anti-tampering techniques, all developed during the ASPIRE project, which will be described in the remainder of this section. Deployment of these techniques on target code is automated via the ACTC, a Python 3 script developed during the ASPIRE project, described in Section 1.1.1.

Two of them, namely *Remote Attestation* and *Code Mobility*, rely on a trusted remote server[28] to verify that the code of the protected application, running on an untrusted machine, has not been tampered with. Therefore the machine running an application protected with such techniques must be connected to the Internet in order for the program to execute correctly. This client-server paradigm [131], has been implemented during the ASPIRE project with an ad-hoc client library, called ASPIRE Client-side Communication Logic (ACCL)[29], and a unified server application, called ASPIRE Server-side Communication Logic (ASCL)[30]. The first contains the methods used by such techniques to communicate with the remote server, and is included in applications protected with on-line techniques. The second one handles requests of the different on-line techniques, within a single server process. The actual client-server communication is done via a custom network protocol, based on the HTTP stack, but hardened in order to resist man-in-the-middle (MITM)[31] network attacks, e.g., avoiding that an attacker tampers a protected application, and then makes the protection techniques to communicate with its version of the server, thus disabling such protections.

The anti-tampering techniques described in the following sections enable the application to identify attempts done by an attacker to modify its code. However, simply identifying that the application has been tampered with is not sufficient, since the final aim of such protection is to obtain an application that, if tampered with, must halt its execution. Thus, if any tampering with the program code is detected, these techniques activate one of the *reaction* mechanisms developed during the ASPIRE project [42]. The latter are able to stop the tampered application execution, either immediately or after a specified time period. However, since the reaction mechanisms code has not been open-sourced, they will not be included in this thesis.

[28] The server is trusted by licensors of the protected application, typically because is under their direct control; for example, for a proprietary application, the server could be located at the software company premises.

[29] https://github.com/uel-aspire-fp7/accl/

[30] https://github.com/uel-aspire-fp7/ascl/

[31] https://www.owasp.org/index.php/Man-in-the-middle_attack

Anti-callback Stack Checks

This technique [48], implemented by DIABLO, identifies attempts by an attacker to call a protected function in points of program execution not originally intended by the application developers. For example, consider a program that handles sensitive data, such for example medical records of patients in a hospital. To prevent privacy breaches, such records could be stored encrypted, with the key for decryption hidden in the program code (e.g., by means of one of the data obfuscation techniques depicted in Section 1.1.2). If a user tries to display a record, the program calls a routine that checks the user credentials, in order to verify that he or she has the authorization to view such records. If this check succeeds, another function is called, decrypting the record and showing it to the user. Thus, an attacker having access to the encrypted record, but without the proper authorization, could try to avoid this check, by calling directly the decryption function. This can be done even without tampering with the program code. The process could be forced to load a malicious dynamic library, with an attack known as Dynamic-Link Library (DLL) injection [72], which will call the decryption function when the program is executed; for example, this could be done by substituting an OS dynamic library, which the target application needs to load, with a version of such library tampered by attacker.

To prevent this attack, this protection technique relies on checks that verify, when a call to a protected function occurs, if the caller function is part of the application code (and not of an attacker injected library); the checks are located in the code at entry points of the protected functions. In practice, these checks read the first position of the call stack, which contains information about the last function call executed by the program. Since checks are inserted at the entry point of the protected function, the aforementioned information will be related to the call made to the protected function immediately before the execution of the checks. By checking the return address, this technique is able to verify if the caller function belongs to the same library of the protected one, by verifying that the return address specified in the call stack is comprised in the memory segment hosting the library code. Therefore, this protection technique is able to ensure that a protected function is called only by other functions of the same library, thus blocking calls by dynamic libraries injected by an attacker.

Remote attestation

Attestation is a technique that aims to verify if the code of a target application has been tampered with. An external application, namely *appraiser*, is responsible for the verification. The latter is carried out by an ad-hoc component of the latter, called *attestation manager*. The verification is executed by evaluating measurable characteristics of the target application execution (e.g., a hash of the memory hosting the target code), and comparing them with the results of an evaluation carried on an un-tampered copy of the application executed on a trusted machine, prior to distributing the software. This process can be repeated multiple times during the execution of the target process, to ensure that dynamic tampering does not occur during the whole lifespan of the process execution. This general architecture of the technique has been proposed by Coker *et al.* in [36].

The generic workflow of software attestation is the following:

1. the appraiser sends an *attestation request* to the manager;
2. the manager calculates the requested measurements, and sends them to the appraiser;
3. the appraiser decides if the target application code has been tampered with, basing such decision on the measurements received from the manager;
4. the appraiser informs the manager of its decision;

5. if the appraiser decides that the application has been tampered with, a reaction procedure (if implemented) can be put in action by the manager, e.g., terminating the application.

Implementations of this technique can be grouped based on whether the measurements are evaluated with dedicated hardware, e.g., leveraging a cryptoprocessor[32] mounted on the machine executing the target application, or with a software implementation, therefore distinguishing between hardware and software attestation [8]. A notable example of the first is the Trusted eXecution Technology (TXT)[33] marketed by Intel, which attests the whole OS code running on the target machine, leveraging a hardware cryptoprocessor, called Trusted Platform Module (TPM) and designed by the Trusted Computing Group (TCG)[34].

A further distinction can be made based on the location of the appraiser code. In local attestation techniques, it is deployed on the machine running the target application, while software attestation techniques require the appraiser to be located on a remote server. An example of local software attestation is the technique known as *code guards* [32], where the appraiser is embedded in the target application. Remote attestation has been for example implemented by Intel[35] via hardware support, in particular with Software Guard Extensions (SGX) [39].

Finally, another categorization of software attestation techniques relates to the characteristics on which the measurements are based. Techniques that take into account only static characteristics of the code, such as for example instructions and constant data loaded in memory, constitute the static attestation category, while the others are generically called dynamic attestation techniques. An example of the latter are the ones based on *software invariants*, dynamic characteristics of the code (e.g., variable values, instructions executed) that remain constant in all possible executions of the application, e.g., regardless of the input given to the latter. A practical implementation has been proposed by Abadi in [2] leveraging *data likely invariants*, which are properties of variables values, such as falling into a defined range, which should be respected during the whole program execution and are evaluated with an empirical analysis of the application execution traces. However, this approach has been evaluated as unfeasible in real-life applications by Viticchié *et al.* in [17].

ESP supports a static remote software attestation technique, described in [124], which permits to specify areas of the target application code that are likely to be tampered with by an attacker. Measurements are calculated at run-time by hashing the memory locations containing instructions of the selected code areas. Various hash algorithms are supported, including Blake2 [10] and SHA256 [46]. As said before, on-line techniques developed in the ASPIRE project use a unified client-server architecture, described at the beginning of this section. Remote attestation fits in this architecture by implementing the appraiser as a sub-routine of the ASCL, while the attestation manager code is placed inside the target application, and uses the ACCL library functions to communicate with the server.

Remote attestation has the problem of being subject to replay attacks. An attacker can record attestation messages sent by an un-tampered version of the application, and then send these messages to the appraiser after having tampered with the program code. This specific implementation solves this problem by correlating the measurement calculation with nonces sent by the verifier with the attestation request. In particular, such nonces will drive a random walk

[32] A cryptoprocessor is a microprocessor (or a System-On-Chip) specifically designed to executing cryptographic operations; typically, it has various physical security measures to resist hardware tampering.

[33] https://www.intel.com/content/www/us/en/support/articles/000025873/technologies.html

[34] https://trustedcomputinggroup.org/

[35] https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing

algorithm [101], in order to define the order by which the instructions memory addresses will be taken into account by the selected hash algorithm; therefore, an attacker should first find the location of the nonce in the attestation request message, and consequently understand how the nonce is related to the hash calculation.

Code mobility

Code mobility is an on-line anti-tampering technique, designed by Cabutto *et al.* [27], which protects an application code from both static and dynamic modifications by moving areas of code to a trusted remote server, called *mobile blocks*, which are deemed sensitive from a security point of view. To harden tampering of the protected code, while preserving the target application functionality, mobile blocks are downloaded on an as-needed basis. If and only if instructions in a mobile block must be executed, the latter is downloaded from the server.

With the current implementation, code blocks with a single entry point are supported. These can be whole functions of the program, or parts of a function control flow, with a granularity level of single basic blocks. Also, moving data sections is not supported by the technique, therefore only instructions are moved, while static data (i.e., constants) are left untouched in the application binary. The architecture of this protection technique comprises the following components:

- the *code mobility server*, implemented as a sub-routine of the ASCL, which delivers mobile blocks when asked by the application running on the client;
- the *binder*, a component in the target application that monitors the execution flow, and detects when a mobile block must be downloaded from the server;
- the *downloader*, another component in the target application that, when tasked by the binder, downloads the needed mobile block from the server, via calls to ACCL functions, and saves it in the program memory, so that the contained instructions can be executed by the client.

Essentially, when the target application is protected with this technique, the blocks removed from the binary are substituted with calls to the binder, which, at run-time, will call the downloader, and then will redirect the program flow to the first instruction of the downloaded block. To further increase the attacker effort in tampering with the protected code, mobile blocks are not downloaded in the code section of the program, but at random locations in the heap.

It should be noted that the binder is called only when instructions in a mobile block must be executed for the first time; downloaded mobile blocks are subsequently left in the client memory. Clearly, the protection could be enhanced by downloading the blocks every time their instructions must be executed, and removing them from memory immediately after their execution. However, this approach has not been deemed feasible since this technique has been developed in the ASPIRE project, which targets specifically Android applications for mobile phones. First, mobile operators offer plans that comprise a limited amount of Internet traffic, so the communication with the server must be limited to avoid the exhaustion of the available traffic that would lead to charges from the service provider to the application user; second, mobile phones have generally limited computational resources, thus calling the binder every time a mobile block must be executed could lead to significant slowdowns of the application; finally, mobile Internet connections are not always reliable, therefore this architectural choice limits the possibility of a target application halting execution due to the impossibility of downloading a mobile code when requested, thus preserving user experience.

1.1.4 Anti-debugging

Debuggers are one of the most common tools used by malicious reverse engineers. Indeed, they can be employed by an attacker for a variety of purposes, both for reverse engineering an application and to tamper with its code. For example, from a reverse-engineering point of view, they can be used to dynamically inspect the behavior of an application at run-time or to collect execution traces that can be analyzed after program termination. For tampering purposes, debuggers may be utilized to halt an application execution, modify its memory locations (in code or data sections), and then resume it, therefore changing the application behaviour for the remainder of its execution. Many debuggers are available, either FOSS or proprietary, the most notable being GDB[36] for Linux applications, OllyDbg for Windows programs[37] and IDA Debugger[38] for both.

Thus, when protecting an application, preventing a debugger from being attached to it would be a desirable outcome. For this purpose, many techniques have been proposed, generically known as anti-debugging protections. It should be noted that, theoretically, the OS could provide such protection, since debuggers rely on system calls to work. For example, on a Linux system, a debugger can be attached to a target process by calling, within the debugger code, the `ptrace`[39] system call, with the following arguments:

```
ptrace(PTRACE_ATTACH,pid,0,0)
```

where `PTRACE_ATTACH` is a constant identifying the attach request, and `pid` is the target process identifier. However, a Linux application that do not want to be debugged can communicate this to the OS by calling the `prctl` system call with the following arguments:

```
prctl(PR_SET_DUMPABLE,SUID_DUMP_DISABLE,0,0,0)
```

In Linux distributions with a kernel version $\geq$ 2.613, this will reset a flag, contained in the `/proc/sys/fs/suid_dumpable` file, which is checked by the Linux OS when a debugger asks to be attached to a process. If the flag has been reset with the aforementioned procedure, the `ptrace` call by the debugger will have no effect, and the debugger will be prevented from attaching to the target process. However, this protection can be easily circumvented by an attacker. If he has root access to the OS, after the target process calls `prctl`, he can simply set the flag again by modifying directly on the `suid_dumpable` file.

Since in a MATE scenario an attacker normally uses his machine to operate on the target program, he will clearly have root access to the OS, therefore this protection is not suitable in this case[40]. However, applications running on untrusted machines can be protected against debugging with *self-debugging*, a technique that is built upon an inherent limitations of all major Operating Systems. A process can be debugged by only one other process. An application protected with this technique leverages this OS constraint by launching, immediately after starting its execution, a custom debugger that will immediately attach the protected process, taking the only spot available to debug the target application. To prevent the attacker from removing the protection debugger and attaching his or her own one, some application logic is moved to the protection debugger, so that the protected process will stop behaving correctly if an attacker detaches the self-debugger from it. A simple way to implement this schema is substituting some

[36] https://www.gnu.org/software/gdb/

[37] http://www.ollydbg.de/

[38] https://www.hex-rays.com/products/ida/debugger/index.shtml

[39] http://man7.org/linux/man-pages/man2/ptrace.2.html

[40] It would be applicable if the target application runs on a secure server, with the attacker having remote access to it, but without root capabilities.

jump instructions in the protected process with debug exceptions. The control will pass to the self-debugger, which, given the interrupt instruction address in the protected code, will resume its execution at the target address of the substituted jump instruction. For example, this implementation of self-debugging has been used to protect the Starcraft II[41] video game. However, also this implementation can be easily circumvented by an attacker, e.g., by statically analyzing the self-debugger, locating the debug exception handlers, and restoring the original jump instructions in the protected application.

ESP deploy a self-debugging technique, developed by Ghent University during the ASPIRE project, and detailed in [3], which overcomes the aforementioned limitations. Using DIABLO, which, as described in Section 1.1.2, is able to rewrite the binary code of the target program, whole parts of the protected application code can be moved to the self-debugger. Every time the program execution falls into a part of code moved in the self-debugger, the protected application throws a debug exception. The self-debugger will copy the target process context to execute the moved instructions, and will be able to read and write the target process memory via the `ptrace` system call, with the first argument set respectively with the constants `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`. The attacker can try to restore the moved instructions in the original position in the target application code, in order to detach the self-debugger and obtain a working application. However this it is not a trivial process. For example, registers used by the instruction are changed when the context is copied by the self-debuggers, and instructions accessing memory are substituted with self-debugger ad-hoc routines that in turn use the aforementioned `ptrace` system calls. It should be noted that this technique has been patented by their inventors [125], but its code[42] license allows free use for non-commercial purposes.

1.2 Knowledge-based and expert systems

Knowledge-based systems are computer programs designed to solve problems by reasoning on existing knowledge pertaining to the analyzed problem. Their typical structure [5] comprises two separate components. A *knowledge base*, containing information about the problem domain, and an *inference engine*, which uses a set of logical rules to reason on existing knowledge, in order to infer new information that will be saved in the knowledge base and can be further reasoned on, ultimately leading to a solution for the given problem. In this way, the knowledge base will be expanded automatically, in a process known as *self-learning*. Finally, a user interface is included in the system to manually add existing information in the knowledge base, for providing the system with problems to solve, and to present the user with the solution of the problem, along with an explanation of the reasoning that led to such solution.

This architecture is typically used to implement *expert systems* [70], computer programs aiming to solve problems by emulating the decision-making process of a human expert in the domain of the problem. Expert systems are built by gathering domain-specific knowledge among experts of the problem field; this knowledge is then formally modelled as structured data saved in the knowledge base, and logical rules constituting the inference engine.

1.2.1 Knowledge bases and ontologies

The knowledge base contains all information that is deemed useful to solve the specific problem. It can be implemented via an *ontology*, a data representation that, as the classic Greek

[41] http://web.archive.org/web/20180303093853/http://www.bhfiles.com/files/StarCraft II/Wings of Liberty (Beta)/0x1337.org - SCII Anti-Debug.htm

[42] https://github.com/csl-ugent/anti-debugging

philosophical discipline with the same name [115], aims to model the information regarding a specific knowledge domain. Abstract concepts, their hierarchical representation, relationships between them, and specific entities that embody the aforementioned abstract concepts. Ontologies are written using formal languages, known as *ontology languages* (e.g., Web Ontology Language 2 (OWL2)[43], RDF Schema[44]) that are able to represent domain-specific information with the concepts of *individuals*, *classes*, *attributes*, *relations*. For example, consider the example ontology about vehicles in Listing 1.5, written in the OWL2 Manchester Syntax[45].

```
Class: Vehicle
EquivalentTo: (wheelsNumber min 1) and (producedBy only Manufacturer)

Class: Car
EquivalentTo: Vehicle and (wheelsNumber exactly 4)

Class: Truck
EquivalentTo: Vehicle and (wheelsNumber min 5)

Class: Manufacturer
EquivalentTo: (manufactures some Vehicle) and (hasNationality some
    Nationality)

Class: CarManufacturer
Equivalent: Manufacturer and (manufactures some Car)

Class: TruckManufacturer
Equivalent: Manufacturer and (manufactures some Truck)

ObjectProperty: produces
InverseOf: producedBy
```

Listing 1.5: Example ontology about vehicles, written in the OWL2 Manchester Syntax.

First, there are two general concepts, or classes. Vehicles and manufacturers. The concept of vehicle is characterized by an attribute, i.e., the number of wheels, and a relationship with the manufacturer of the vehicle. Attributes represent specific properties of a class, while relationships bind together different concepts. The class scheme is hierarchical, so that for example the concept of car is a specialization of the general concept of vehicle, in particular representing that a car is a vehicle with four wheels. Classes, with their attributes and the relationships between them, fall under the term *axiom*, and in general describe the structure of the model. Specific instances of the abstract classes, representing the data described by the aforementioned model, are represented with *individuals*, for example the ones in Listing 1.6.

In a certain sense, an ontology may be defined as a database, with the axioms being the database schema and individuals corresponding to the information populating the database. However, ontologies and databases on how information is treated. First, they treat differently missing information when elaborating queries. In the example, given that information about the nationality of Volvo manufacturer is not given, a query in the form "Volvo is Swedish?" will be negatively answered by a database, since the latter treat missing information as false, while the ontology will give an uncertain answer. Another difference is that database schemata can only constraint

[43] https://www.w3.org/TR/owl2-overview/

[44] https://www.w3.org/TR/rdf-schema/

[45] https://www.w3.org/TR/owl2-manchester-syntax/

```
Individual: "Testarossa"
Types: Car
Facts: hasManufacturer Ferrari

Individual: "Volvo FH13"
Types: Truck
Facts: hasManufacturer Volvo

Individual: "Ferrari"
Types: CarManufacturer
Facts: hasNationality Italian

Individual: "Volvo"
Types: Manufacturer
```

Listing 1.6: Individuals in the example ontology about vehicles.

the data (e.g., that the number of wheels is a positive integer), while ontology axioms can treat implicit information, e.g., is not necessary to specify the number of wheels of a Ferrari Testarossa, since it is defined as a car, which in turn is a vehicle with four wheels.

1.2.2 Inference engines

As already said, an inference engine is a component of a knowledge-based system, able to reason on the information contained on the knowledge base to infer new information. To do so, inference engines apply to existing information a set of human-defined logical rules, in order to reach a goal, such as evaluating a user assertion as true or false. Rules are formally expressed as if-then rules, such as, referring to the vehicle ontology in Section 1.2.1, "Rule1: X is a manufacturer AND X is Italian ⇒ X is an Italian manufacturer". Formally, a rule is expressed as a Horn clause [65], which has the following form:

$$\text{Rule name: premises} \Rightarrow \text{consequence}$$

Which entails that, if the premises, containing a logical combination of basic sentences, evaluate true, then the sentence also evaluates to true. Inference engines and their rules are typically written using ad-hoc logic programming languages. A notable example is Prolog, developed in 1972 by Alain Colmerauer at the Aix-Marseille University, formalized by ISO in 1995 [67], and available at time of writing in various implementations, both free (e.g., GNU Prolog[46], SWI-Prolog[47]) and commercial (e.g., LPA-PROLOG[48], SICStus[49]).

Inferencing on such rule can be implemented using either forward or backward chaining (or reasoning). Forward chaining tries to reach the user-defined goal from the available information, searching for a chain of rules that in the end reaches the goal. Instead, backward chaining starts from the goal and tries to find a chain that ends with rules that are verified by axioms in the ontology. For example, considering the following set of rules regarding the vehicle ontology:

1. Rule1: X is a manufacturer AND Y is a nationality AND X has nationality Y ⇒ X is a manufacturer of nationality Y

[46] http://www.gprolog.org/

[47] http://www.swi-prolog.org/

[48] http://www.lpa.co.uk/

[49] https://sicstus.sics.se/

2. Rule2: X is a manufacturer of nationality Y AND Z is a vehicle AND X produces Z ⇒ X is a vehicle of nationality Y

3. Rule3: X is a vehicle of nationality Y AND X is a car ⇒ X is a car of nationality Y

Suppose that the inference engine is asked to verify the proposition "Testarossa is a car of nationality Italian". With a forward chaining approach, the goal will be reached verifying the first rule, obtaining that "Ferrari is a manufacturer of nationality Italian" from axioms in the ontology, then the second rule, asserting that "Testarossa is a vehicle of nationality Italian" both from the consequence of the first rule and from axioms, and finally the third rule, reaching the goal by verifying the sentence "Testarossa is a car of nationality Italian" supplied by the user. With a backward chaining approach, the inference engine would start from the goal, see that it is a consequence of the third rule, and then try to verify its premises, checking both axioms and consequences of other rules, stopping when the search leads to all axioms in the ontology. Thus rules would be verified in the inverse order of the forward chaining approach.

1.2.3 Expert systems

As defined before, expert systems are knowledge-based systems whose objective is solving a problem by imitating the reasoning that would be made by an expert in the field of the problem when trying to solve the latter. Information is gathered among experts in the field of the specific problem that the expert system is designed to solve. Generic domain information is saved in the knowledge base, while rules used by the inference engine are designed to mimic the mental processes of the experts when tackling the targeted problems.

The first expert system has been designed and implemented by Buchanan, Feigenbaum and Lederberg in 1965 at Stanford University, and was called DENDritic ALgorithm (DENDRAL) [55]. Its objective was to help organic chemists to discover new organic molecules. The system was designed by both gathering and formalizing generic knowledge of chemistry (e.g., concepts of atoms, molecules, mass) but also modelling the mental processes of organic chemist when tackling the problem, such as for example how the interpreted the results of analysis such as mass spectrometry. DENDRAL was written in the LISt Processor (LISP) functional programming language, which at the time was the standard language for artificial intelligence. Techniques developed in the DENDRAL project were then used to develop expert systems for various purposes. Notable examples are MYCIN [111], designed to identify the bacteria responsible for an infection by asking closed questions to a physician, and eXpert CONfigurer (XCON) [129], which assisted customers of Digital Equipment Corporation (DEC) in buying the most suitable Virtual Address eXtension (VAX) computer system basing on their requirements.

In [63], the author presents a categorization of expert systems based on the type of problem solved:

- *Control* of systems behaviors by interpreting their output (e.g., the INCO expert system [94] used by NASA in Space Shuttle mission control);
- *Debugging* systems, analyzing malfunctions in them and prescribing remedies;
- *Design* of objects under constraints (e.g., DENDRAL [55]);
- *Diagnosis* of system malfunctions from observable behaviors of the system (e.g., MYCIN [111]);
- *Instruction* problems, analyzing student behaviors and proposing learning methodologies better suited to specific students;

- *Interpretation* of sensor data, inferring situation descriptions (e.g., XCON [129]);
- *Monitoring* of systems, comparing observations gathered over time to identify vulnerabilities;
- *Prediction* of likely consequences of given situations;
- *Planning* of actions needed to reach a goal;
- *Repair* a system, executing a plan to administer a prescribed remedy.

In the field of cybersecurity, research on the possible application of expert systems for risk analysis of computer network and systems has been active from 1986 at least. In [64], Hoffman suggests that, even if at the time expert systems were generally in an early stage of development, using them for risk analysis might lead to a positive outcome, especially after building a suitable general model for cybersecurity experts knowledge. The author theorizes that an expert system built in this way would be able to identify vulnerabilities in the analyzed computer system configuration and suggest the appropriate countermeasures to reduce the overall risk. In the same year, Denning and Neumann started the development of the Intrusion Detection Expert System (IDES) [49], an host-based Intrusion Detection System (IDS) mixing an expert system with statistical anomaly detection techniques to detect unauthorized access, both by local and remote users, of resources hosted by the monitored system. A first prototype, monitoring a DEC-2065 computer running the TOPS-20 operating system, has been presented in 1990 and is detailed in [87]. An evolution of the aforementioned system, renamed Next-generation Intrusion Detection Expert System (NIDES) and detailed in [6], and supported real-time analysis of inter-process communications for host security purposes. Another IDS expert system developed in the same years was the Network Intrusion Detection eXpert system (NIDX), presented in [18], targeting UNIX System V machines and able to suggest to a network administrator possible security breaches, with a knowledge base that contains information about the structure of the monitored network, and usage profiles of its users, thus mimicking the decision process of administrators. Systems with similar capabilities have been implemented in the same period. The Network Anomaly Detection and Intrusion Reporter (NADIR) [69], to monitor the internal network of the US Los Alamos National Laboratory, and Haystack [113], to detect breaches in US Air Force systems. In particular, the latter presented self-learning capabilities, being able to evolve user profiles over time. Such changes are summarized to the system administrator so that he or she could assess such evolution in order to verify that breaches are not masked by this system behaviour. All the aforementioned systems aimed to detect security breaches at real-time. Instead, the Expert System for security Auditing (AudES) [119] was developed to assist computer security auditing process, such as the analysis of log-in or resource access records after a security incident.

More recent research on network security expert system has focused on enhancing breach detection performances of such tools by combining the classic architecture of expert systems with other techniques. For example, Eronen and Zitting [53] present a tool to analyze firewall rules. The network administrator can write author's knowledge-level rules using constraint logic programming [71], which are subsequently translated by an expert system into a low-level access control list written in the format used by Cisco routers[50]. Also, combinations of machine learning approaches and expert systems have presented. Examples are [50] by Depren *et al.*, using Self-Organizing Maps (SOM) and decision trees for breach detection and an expert system to interpret the result of such machine learning algorithms, and [100] by Pan *et al.*, which uses neural networks for detecting attacks leveraging unknown vulnerabilities, while an expert system identifies known attacks. Fuzzy logic algorithms have also been embedded in expert system for real-time intrusion

[50] https://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html

detection, as in [98], and post-incident network forensics, for example in [77] and [84]. Finally, no applications of expert systems technology to the software protection field has been found in literature. This thesis aims to fill this gap in research, presenting an expert system with the objective of detecting vulnerabilities in applications, and suggesting to the software developer suitable protections to mitigate such vulnerabilities.

Chapter 2

Decision support system for software protection

> In any moment of decision, the best thing you can do is the right thing, the next best thing is the wrong thing, and the worst thing you can do is nothing.
>
> Theodore Roosevelt

This chapter contains a bird's eye view on ESP, providing general information about how, starting from a vanilla application, its components, processes and data are orchestrated to produce a protected version of the application in an (almost) automated fashion. This chapter is organized in the following sections:

- Section 2.1 lists the main requirements for ESP;
- Section 2.2 contains an high-level description of the ESP workflow;
- Section 2.3 presents the results obtained executing ESP to protect an example application (Sumatra, a comparison tool for DNA sequences);
- Section 2.4 details how software security experts validated, during the ASPIRE project, the results produced running ESP on three real-life use cases, summarizing the contents of the D1.06 ASPIRE deliverable [13]; furthermore, the section presents an experimental assessment of the ESP workflow, executed on three different applications of increasing complexity and size.

2.1 Problem statement

The main objective of this thesis is to present a decision support system for automatic software protection, called Expert system for Software Protection (ESP). As detailed in the introduction, protecting software has been a task reserved to few experts in the field, which manually analyze the application to protect, identify the possible vulnerabilities, and then decide the best mitigations, i.e., software protection techniques, to solve them. In doing so, experts base their decisions mainly on their knowledge in the field, their past experience, and ultimately on their instinct. Thus, a decision support system that wants to automatically protect applications needs to mimic

how security experts do their job, following the classical approach of expert systems detailed in Section 1.2, already applied to other aspects of cybersecurity, e.g., network vulnerability analysis and mitigation [6, 18, 69, 119]. To the best of the author's knowledge, no prior works in literature apply the expert system methodology in the software protection field.

The software protection process may be more formally defined framing it as a risk management process, a customary activity in various industry sectors, such as finance, pharmaceutics, infrastructure, energy and, of course, information technology. Regarding the latter, the National Institute of Standards and Technology (NIST) standard for information security [68] outlines a high-level workflow for IT systems risk management, comprising the following phases:

1. *risk framing*: establish the scenario in which the risk must be managed;
2. *risk assessment*: identify threats against the system assets, vulnerabilities of the system, the harm that may occur if such vulnerabilities are exploited, and the likelihood this would happen;
3. *risk mitigation*: determine and implement appropriate actions to mitigate the risks;
4. *risk monitoring*: verify that the implemented actions are effective in mitigating the risks.

2.1.1 Risk framing

To frame risk, it must be taken into account that this thesis targets MATE scenarios, with software running on untrusted machines, therefore an attacker has complete access to the software binary code, and can monitor its execution. A broad range of tools (e.g., debuggers, decompilers, disassemblers, fuzzers) can be used by an attacker to endanger the software assets, whatever its drive may be (e.g., financial gain, industrial espionage, or simply personal satisfaction). However, a less experienced attacker may not be able to use effectively these tools, therefore more complex attacks may be executed only by few attackers. Conversely, more experienced attackers may not be interested in attacking the application that must be protected. For example, a seasoned attacker is unlikely to spend time in hacking a low value application, thus protections against more complex attacks may be unnecessary, or even unfeasible from a financial point of view. Spending thousands of dollars to safeguard an application licensing scheme, acquiring licenses of strong proprietary protection schemes, is financially sustainable only if the value of the application and the expected turnover from the application sales justify it. Thus, different attacker profiles, with increasing levels of skill and experience, must be modelled, so that ESP can take this information into account, in order to correctly assess risk depending on the application value.

Also, as presented in the Introduction, ESP must be designed for two different usage scenarios. First, ESP can be useful to application developers, without specific knowledge in software security, which do not have the financial resources or the will to hire security experts to protect their application. In this case, ESP must implement a completely automated workflow, driving a set of automated protection tools (as the ones presented in Section 1.1.1), able to provide the software developer with a protected application binary, limiting as most as possible the interaction required with him or her in the process. Second, ESP can constitute an interesting tool for security experts, which want to obtain a first solution to the problem, which they can analyze to assess its effectiveness, and further manually refine if necessary. Thus, insights on the results of all the workflow phases must be provided.

Finally, as pointed out by security experts interviewed in the ESP design phase, the target application life-cycle must be taken into account. Prior to software release, enough time is available to carry out a thorough assessment of the application risks and to infer and apply the possible mitigations. However, after distribution of the application, possible bugs may surface. After users encounter such problems and report them, a patch that solves these issues must be released as

soon as possible. Thus, the amount of time to find possible threats or vulnerabilities introduced in the new version of the application by the aforementioned patches is limited, and a more shallow assessment of risk must be performed. Therefore, ESP must provide different usage profiles. Accuracy of results produced by its reasoning processes will depend on the time available to perform them. Thus, the user can choose the more appropriate profile, given the software life-cycle phase in which ESP is employed.

2.1.2 Risk assessment

Regarding risk assessment, a first fundamental activity is defining what is an asset in software security, and which security requirements it may have. A software asset is any code or data comprised in the application that, if successfully attacked, involves a damage for the software proprietors, for example financial losses (e.g., lost sales due to an unlicensed version of the application illicitly distributed) or reputation damage (e.g., disclosure of sensitive data of the application users). To model assets, specific asset security requirements, i.e., asset properties that must be safeguarded against attacks, must be defined, in order to infer possible attacks against them and the appropriate mitigations. Following [68], two possible security requirements[1] are defined:

- *confidentiality*: preventing the disclosure of parts of the application that must remain unintelligible to the attacker, such as proprietary algorithms or cryptographic keys;
- *integrity*: preserving code that must not be altered by the attacker, for example licensing schemes.

ESP needs to include a risk assessment phase, where possible threats against the application are identified, given a set of asset security requirements specified by the application developer. Requesting this information from the latter has been deemed reasonable by experts interviewed in the design phase of ESP, and, to the best of the author's knowledge, there is no prior work in literature automatically identifying assets. To ease developers in this preliminary task, a method must be devised to include requirement definitions in the source code, so that this may be done during the application development and not with an a-posteriori analysis, which can be cumbersome and error-prone, especially with complex applications.

Then, ESP needs a way to assess risk. Following the classic structure of expert systems, ESP must include a knowledge base, containing all the information needed to correctly infer the threats against assets, and an inference engine, which must model the attacker mental processes in devising actual attacks having as target the breaching of at least one asset security requirement. Also, we need a way to represent the threats against application assets found with the inference engine. A well-known approach in literature to threat modelling is the use of *attack graphs*, first proposed by Swiler *et al.* [117] for computer networks risk assessment. Each threat is modelled with a tree, having as root node the target of threat (in the ESP case, an asset security requirement), while other nodes correspond to indivisible *attack steps*, simple attacker actions (e.g., executing the application, debugging a function); each attack step has a set of preconditions that must hold for the attacker to execute it (e.g., to debug a function, a debugger must be attached to the application) and a set of effects if the attacker succeeds in carrying out the task (e.g., debugging the

[1] The NIST standard identifies an additional requirement, *availability*, defined as "Ensuring timely and reliable access to and use of information.", which is applicable for a network scenario, for example hardening the system against DDoS attacks. However, in a MATE scenario, this requirement can be safely ignored, since attacks on the target application remain confined to the attacker machine, not endangering the application availability for legitimate users.

function, the attacker understands its business logic, or finds values of local variables at runtime). Starting from the root of the graph, *attack paths* can be inferred, i.e., ordered sequences of specific actions that an attacker may execute to substantiate its threat against an asset, breaching a specific security requirement of the latter. Thus, to implement an inference engine able to build attack graphs for each application asset, a set of simple tasks that an attacker may execute in a MATE scenario must be enumerated (e.g., analyze execution traces, find a function in the binary code), associated with a set of preconditions and results, in order to obtain meaningful paths. The attack steps level of detail must be also set carefully, so that attack steps are neither too vague or unnecessarily specific: on one hand, using too much generic attack steps can lead to paths lacking expressiveness, with the risk of overlooking possible mitigations, while on the other hand, too detailed attack steps would lead to an impractical high number of equivalent attack paths, which could be solved with the same mitigations, with a detrimental effect on the performances of the inference engine, and increasing the effort needed to manually analyze the inferred attack paths, if the ESP user wants to assess them prior to the mitigation phase. Thus, this ESP phase must be validated: the level of detail of attack steps must be tuned accordingly, by evaluating the accuracy of automatically inferred attack paths on test applications in describing actual attacks executed by security practitioners against such applications.

Also, relationships between generic attack steps and the application structure must be defined, to model realistic attack steps (and consequently paths) that represents the real actions that an attacker must execute to endanger the analyzed applications: for example, if an attacker wants to change the initialization value of a local variable of a function, he may first locate the latter in binary code, and then focus its analysis on the function code to locate the specific variable and subsequently tamper with the initialization code. Finally, ESP must be able to quantify the probability of an attacker succeeding in the attack must be assessed, taking into account the attacker skills, but also the application code characteristics: for example, the effort needed to understand the logic of a function increases with its complexity, in terms of number of instructions comprised in the function code, amount of nodes, edges and possible paths in the CFG, and number of different memory locations accessed by the instructions. This leads to another requisite for ESP: it must be able to assess the complexity of code constituting application assets. Abundant research has been done in the field of software engineering regarding this issue, with the definition of various software metrics devised to analyze various aspects of code, in order to obtain quantitative measures that summarize its complexity: notable examples are metrics proposed by Halstead [62] and McCabe [88].

2.1.3 Risk mitigation

Given the threats modelled in the risk assessment phase, i.e., the attack paths against the security requirement, ESP must be able to infer the possible mitigations against these requirements. Indeed, software protections, as provided in the overview in Section 1.1, are designed to safeguard the security requirement of assets, countering possible attacks against them. However, it should be noted that in a MATE scenario, an attacker with sufficient motivation and resources will be ultimately able to find a way to remove or circumvent protections, thus succeeding in breaching such security requirement: in other terms, there is no perfect software protection, as it has been formally proved for obfuscation techniques by Barak *et al.* in [11]. However, protections may be still effective, if they succeed in deferring an attacker (given its profile, as defined in Section 2.1.1) for a sufficient amount of time: for example, protections safeguarding a license check embedded in a proprietary application may be deemed effective, if the software remains un-cracked for a time sufficient to provide an economic return (i.e., applications sales) that justifies the previous investments in the application developing. So, the objective of ESP, targeting a MATE scenario, is to infer a set of protections that must be used to safeguard the application assets, deferring for

the longest time possible the substantiation of threats against them, i.e., successful attacks.

Therefore, ESP must be able to find, for each attack path inferred in the risk assessment phase (see Section 2.1.2), a set of possible mitigation that can be effective in defending the asset endangered by the threat. In other terms, each attack path against one or more security requirement of an asset must be countered by applying the appropriate protections to asset code. To do so, a mapping between the available protection techniques and the security requirements that such protections are able to safeguard is indeed useful: however, it is not sufficient, since the objective is deferring a possible attacker for as long as possible. This involves defining a method to quantify the effectiveness of protections able to counter the attack paths, in order to compare possible mitigations and infer the ones among them that are most suitable to protect the analyzed application against assessed threats. The concept of *potency* of transformation, introduced by Collberg *et al.* [37], is a good starting point to solve this problem, since it binds protection effectiveness with software metrics of the asset code on which the protection is applied, quantifying it with the increase of metrics in the protected application w.r.t. the original one: in the aforementioned work, Collberg *et al.* propose a set of metrics that can be used to evaluate the efficiency of code obfuscation (see Section 1.1.2), while more general metrics are proposed by Tonella *et al.* [118]. Thus, building on this prior research, software metrics appropriate to assess the real effectiveness of protection techniques, evaluating their performance when applied on the specific application assets that must be protected; furthermore, such metrics must be evaluated by ESP in an automated fashion, analyzing the source code of the target application.

It should be also noted that applying protections to the application code has the side-effect of slowing it down. An application is useless if it is well protected but too slow to appeal to the user. The ESP mitigation process will take this problem into account, aiming to find an optimal trade-off between the achieved security for the assets, and the performance overhead introduced by the selected mitigations. Overhead can clearly be assessed after protecting the application, executing a previously instrumented protected binary, and measuring the additional time, memory and network bandwidth needed to execute the application. However, this execution test should be done automatically: this would require the developer to provide a set of tests for the application, and also would be problematic if the target program has a Graphical User Interface (GUI); furthermore, the application should be tested for each possible set of mitigation inferred by ESP, and this would lead to unfeasible times, especially in the case of big applications, with many assets and a non-negligible time for execution. Thus, a method to approximate the introduced performance overhead on the analyzed application, without actually executing it must be devised: for this purpose, the use of software metrics for overhead estimation will be investigated.

Also, as detailed in Section 2.1.1, ESP must be able to drive a set of automated protection tools, in order to protect the code without requiring user intervention. To do so, the information needed to operate such tools must be modelled. First, tools are able to enforce various protections (e.g., DIABLO supports different code obfuscation techniques, as reported in Section 1.1.2), so each protection technique must be mapped to the tools enforcing them. Then, for each specific implementation of a technique provided by a tool, all the necessary configuration parameters for the latter must be identified: this is necessary also to infer realistic mitigations, since these parameters may influence the obtained security level and performance overhead after using a tool to protect an asset. For example, DIABLO provides a parameter to specify the percentage of basic blocks that must be obfuscated: clearly, setting a high value for this parameter will result in a thorough obfuscation, but will also incur in a considerable slow-down when the obfuscated code is executed. So, different usage profiles for these automated protection tools must be modelled, so that ESP can take them into account in its reasoning processes, and can subsequently drive the automated protection tools with the appropriate values for their parameters.

Another characteristic of mitigations that must be modelled is that interactions among different protections applied to the same assets must be modelled. As emerged in talks with security

experts in the ESP design phase, some protections are able to amplify the effectiveness of other, when applied to the same code: for example, if a local variable is protected with Tigress data obfuscation technique (see Section 1.1.2), the variable value is stored in an encoded form, and decoded on the fly when the variable must be accessed; obfuscating the decoding instructions can certainly increase the effort needed by the attacker to comprehend the encoding schema and compute the real variable value. Conversely, there are cases of incompatibilities among protections: for example, after protecting a function with remote attestation (see Section 1.1.3), further modifications cannot be made to the protected code, since this would be erroneously identified by the attestation technique as a tampering attempt by an attacker: so, protecting the same code with remote attestation, and subsequently for example with CFF (a code obfuscation technique described in Section 1.1.2), leads to a non working application; however, the inverse order of application of protections is perfectly legal. So, ESP must infer also the order of protections that must be applied to the code, trying to leverage the aforementioned "amplification effect" to improve the protected application security level, while illegal sequences of protections must be avoided.

Finally, there is another side effect of protections that must be considered by ESP. Some protections expose a "fingerprint", an introduced code structure or run-time behavior that can be recognized by the attacker, when he or she analyzes the protected binary: thus, an attacker may look for these fingerprints introduced by protections, easily finding the protected assets in code, and thus concentrating its effort on a more limited amount of code. Examples may be the large CFG introduced by CFF, or the increased network activity of an application protected against tampering with code mobility (see Section 1.1.3). This problem was highlighted by security experts involved in the preliminary design phase of ESP. The solution to this problem adopted in software security companies is obfuscating as much code as possible, applying this protection even on areas of code not sensitive from a security point of view (i.e., non-assets), and then executing the application to assess its performances: if the additional obfuscation introduced too much overhead, the cycle is repeated, diminishing the areas of application of the technique and testing again the program: indeed, this is a tiresome manual process, which involves potentially multiple cycles of obfuscation and testing, until assets are hidden from the attacker with acceptable application performances. So, a set of strategies must be devised and implemented in the ESP risk mitigation phase, in order to limit this side effect of protections, finding ways to hide assets and confuse the attacker when he or she analyzes the target program; also, having an automated asset hiding phase would be an appealing plus for software security experts using ESP.

Summarizing, the ESP risk mitigation phase should propose a comprehensive *protection solution*, an ordered set of protections that must be applied to the application to increase the effort needed by the attacker to implement the threats found in the risk assessment phase; application of each protection must be detailed with the specific assets targeted and the specific automated tool that must be used to enforce the protection, with the values for the parameters requested by the tool. Also, each solution must be associated with a score, indicating its effectiveness in mitigating possible threats, so that solutions can be compared with each other, and a *best solution* can be inferred and suggested to the user.

2.1.4 Risk monitoring

Monitoring the risk of software in a MATE scenario means essentially constantly verifying that the application has not been hacked after its distribution: this is indeed a complex matter, involving for example monitoring torrent distribution websites for unauthorized redistribution of the protected software; however, this is a task that goes beyond the scope of this thesis. However, software security companies, prior to distribution of a protected application, test its resilience against possible threats on the fields: *tiger teams*, composed by highly skilled white-hat hackers, execute a *penetration testing* phase [7], trying to breach the assets security requirements,

impersonating a possible attacker. The objective of this phase is assessing the security introduced in the software with the mitigation phase: if attacks carried out in this phase are successful, exposing security flaws overlooked by the chosen mitigations, the appropriate corrective measures are taken, with a new mitigation phase aiming at countering possible attacks found by penetration testers. ESP aims to mimic the security experts methodologies: while a real execution of the attack paths found in the risk mitigation phase as been deemed not feasible, since it would require a way to automatically write scripts driving real attack tools to test the protected application. However, the idea of testing the found solutions with the attack paths found in the risk mitigation phase can be applied to assign a more realistic score to inferred protection solutions: since attack steps must be associated with a probability of success, dependent on the attacker profile and the assets code complexity metrics, and since deploying protections to code increase such metrics, the score of inferred protection solutions can be bound to their effectiveness in reducing the probability for an attacker to succeed in the attack paths found in the risk assessment phase, reflecting ESP main objective, i.e., increase the time and effort needed by the attacker to successfully implement its threats against the asset security requirements.

2.2 Automated workflow for software protection

This section outlines the design of the automated workflow for software risk management, highlighting how its phases verify the requirements described in Section 2.1. Furthermore, this section describes the research problems encountered in the design and implementation of each workflow phase, along with the adopted solutions, presenting the main contributions of this thesis w.r.t. the existing state of the art of software protection in the MATE scenario.

2.2.1 Software protection meta-model

Indeed, the main objective of this workflow is automating the software protection process, a task that until now has been reserved to highly trained professionals, with a deep knowledge in the field of software security: due to their experience, they know which vulnerabilities can be exploited by attackers to target an application assets, how to find possible vulnerabilities on the source code, and the mitigations that are better suitable to defer possible attacks. During the ASPIRE project, security experts of the industrial partners of the project have been interviewed, gathering a large amount of information on their *modus operandi* and the background knowledge and experience on which their decisions are founded, when tasked with the protection of an application. Interestingly, while a plethora of security models for computer networks have been defined, to the best of the author's knowledge literature lacks works on software security modelling. Thus, the first contribution of this thesis is a software security meta-model, formalizing the information gathered during the ASPIRE project among experts in the field. All concepts relevant to the software risk management process are modelled, such as attacks, protection techniques, assets and security requirement, with relationships among them: these concepts aim to grasp the background knowledge of security experts, which they put into action when the best mitigations for a given application risks must be decided. Such abstract concepts can be instantiated into formal descriptions of actual protection techniques and threats. Indeed, this is an advancement in description of software vulnerabilities and threats, which, as in the case of well-known databases such as the Common Vulnerabilities and Exposures (CVE)[2] and Common Weakness Enumeration (CWE)[3], are mainly based on free text descriptions of actual threats against specific applications,

[2] https://cve.mitre.org/

[3] https://cwe.mitre.org/

unusable by a machine to automate risk management processes. However, the meta-model goes beyond the modeling of generic software security concepts, formalizing also information about the application that must protected, used by the experts to adapt their general knowledge for the specific task: to model this mental process, a set of relationships have been introduced in the model to link generic security concepts with the actual application structure. Furthermore, the model is able to formalize all the results of the experts work, with a formal structure for possible attacks, in the form of ordered sequences of simple attacker tasks, and of the combination of mitigations that must be applied to the target software to block them, expressed as ordered sequence of actual implementation of protection techniques, stating the tools that must used to enforce them, along with the configuration parameters value to drive the results of their execution. This meta-model can be used by anyone tasked with the protection of an application, in order to organize all the information involved in this process. Also, modelling all the background knowledge and specific application information used by a software security expert, and also the results of its mental processes, the meta-model is also suitable to structure the knowledge base of ESP, which in fact must mimic the expert behaviour. The abstract meta-model, and its implementation as a Knowledge Base for ESP, is described in Chapter 3, which is a reworked version of the publication "A meta-model for software protections and reverse engineering attacks" [15], in which the meta-model has been first presented.

A requirement for all risk management phases is that the target application structure should be taken into account: this is necessary, as described in Section 2.1, so that ESP, carrying out these phases, can produce realistic results (i.e., attack paths, protection that defer them, and the best protection solution to globally safeguard the application). As already said, the meta-model is able to formalize the information about application structure, with a set of classes described in detail in Section 3.2, comprising information about functions, local variables, assets with the user-defined security requirements, but also relationships among them, including a representation of the application call graph and data dependency graph. However, since ESP must implement an automated workflow, all these abstract concepts must be instantiated to describe the structure of the target application limiting as much as possible user intervention. Thus, the ESP workflow includes a preliminary phase, in which the application source code is analyzed in order to instantiate all the related meta-model classes in the ESP Knowledge Base: to do so, ESP first obtains an Abstract Syntax Tree (AST) [79], i.e., a formal description of the application source code, which in turn is analyzed to instantiate the needed meta-model classes in the Knowledge Base. The AST, in ESP implementation, is obtained using Eclipse C/C++ Development Tooling (CDT)[4]. However, assets and security requirements must be marked by the ESP user: a set of code annotations, described in Section A.3 and based on the GNU is Not Unix (GNU) C Compiler (GCC) Attribute Syntax[5], have been defined in order to ease this task for the user; such annotations will be automatically parsed by ESP, instantiating in the Knowledge Base related meta-model classes.

The ESP Knowledge Base has been implemented with an OWL2 [99] ontology (see Section 1.2.1), containing all the modeled background information gathered from the software security experts. ESP manipulates the ontology, inserting additional data produced in the various phases of its workflow, via an external ontology management API[6].

[4] https://www.eclipse.org/cdt/

[5] https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html

[6] https://github.com/daniele-canavese/ontologies

2.2.2 Risk assessment phase

Given the formalization of generic background knowledge of software security experts, and after the modelling of the application code structure described in the previous section, ESP is able to assess the possible threats against the security requirements of application assets defined by the user; in doing so, the reasoning processes used by ESP in this phase must satisfy the requirements detailed in Section 2.1.2.

First, possible threats are formalized in the meta-model using *attack paths*, i.e., ordered sequences of simple and indivisible attacker tasks, called *attack steps*, which an attacker can execute in succession to breach an asset security requirement. Each step is characterized by a set of preconditions: if these are satisfied, the attacker can execute the step. Conversely, attack steps can have postconditions, results of their successful execution by the attacker: thus, a step can enable another one, following in the path, if the second preconditions are satisfied by the postconditions of the first. For example, an attacker, to tamper with a variable at run-time, must be first able to locate it in memory. Also, preconditions can be related to the characteristic of application code, thus binding the inference of possible attack paths with the code structure, satisfying one of ESP requirements for the risk assessment phase: taking the previous example, if the attacker needs the location of a variable at run-time, in order to tamper with its value, he can clearly analyze the program memory searching for the variable; however, the attacker can take another approach if the case of a local variable, first locating the code of the function comprising the variable (e.g., debugging the code), and subsequently analyzing the function instructions, in order to narrow the search to the memory addresses accessed by such instructions. Summarizing, attack paths will be ordered sequences of attack steps, with each of the latter fulfilling with their results the preconditions of following steps: paths will be concluded with a last step, having as postcondition the breaching of the asset security requirement targeted by the path.

Indeed, attack steps represent the execution of an attack task to a specific code or function of the application: information about these tasks, including preconditions, postconditions are consequently necessary to assess the risk. A set of generic attack tasks has been obtained from the study by Ceccato *et al.* [31]: the authors manually analyzed a set of penetration testing reports, executed by professional security practitioners, building a taxonomy of attacks in the MATE scenario. Also, as defined in the risk assessment requirements, the probability of an attacker being able to execute specific tasks has been taken into account: given a set of attacker profiles, with increasing skills and experience (*geek, amateur, professional, guru*), software security experts involved in the ASPIRE project indicated, given their past experience, probabilities relating each attacker profile with each type of attack step: this information has been included in the meta-model, as experts background knowledge, and is used by ESP in its mitigation phase.

Information about attack step types, preconditions and postconditions have been modelled as Horn clauses (see Section 1.2.2): the inference engine uses these clauses to infer the possible attack paths against the assets security requirements, with an algorithm based on backward chaining. ESP can query the inference engine to obtain a complete *attack graph*, i.e., all the possible attack steps having as target the security requirement of a specific asset in the application: the resulting attack paths are stored in the Knowledge Base, formalized with the meta-model classes presented in Section 3.3, and will be the starting point for the subsequent risk mitigation and monitoring phases of ESP. The time needed to execute the risk assessment phase, and subsequently the accuracy of the inferred attack steps, can be limited by the ESP user, setting a hard time limit, or specifying a maximum length for the inferred attack paths. Thus, the execution of the ESP risk assessment phase is adaptable to the application lifecycle, respecting the related requirement defined in Section 2.1.1.

2.2.3 Asset protection phase

The main objective of ESP is, given an application that must be protected and a user-defined list of security requirements for each asset, present the user with the best *protection solution*, i.e., the combination of mitigation able to defer for the longest time possible the attack paths against the application assets, found in the risk assessment phase presented in the previous section. Thus, given such attack paths, ESP can carry out the asset protection phase, with the aforementioned final objective. In doing so, both the risk mitigation and monitoring phases of the risk management process, described in Section 2.1, are carried out: first, the best mitigation for each asset security requirement, taken in isolation, are inferred; then, such mitigations are combined, with each generated combination safeguarding all the attack paths inferred in the risk mitigation phase; finally, a score to each combination is assigned by testing it against the aforementioned attack paths, thus mimicking the penetration testing activity described in Section 2.1.4. The asset protection phase, detailed in Chapter 5, must therefore satisfy the requirements listed in Sections 2.1.3 and 2.1.4.

As already introduced in Section 2.2.1, in order to drive the automated protection tools (Tigress, DIABLO and the ACTC, presented in Section 1.1.1), a set of usage profiles has been defined, specifying for each supported protection techniques, all the configuration parameters that influence the actual application of such techniques. Regarding these, two fundamental concepts are modelled (see Section 3.3 for details): the PI and the Deployed Protection Instance (DPI). A PI represent a specific implementation of a technique, stating the tool used to deploy it automatically on the code, and all the configuration parameters needed by the tool. A DPI represents the actual deployment of a specific technique implementation to a specific variable or section of code in the application (e.g., an asset).

Each protection technique has been assigned a generic level of effectiveness against each of the attack steps types defined in the previous section, obtained by the security experts involved in the ASPIRE project. Furthermore, each protection technique has a specific set of constraints w.r.t. the structure of the asset on which they must be deployed: a simple example is that the Tigress literal and variable obfuscation techniques (see Section 1.1.2 can be applied to the respective types of datum. However, also specific implementation issues of the automated protection tools are considered: for example, the code mobility anti-tampering technique can be applied only to whole functions (and not to a subset of them), due to a limitation of the specific implementation used by ESP. Checking this constraint, ESP is able to infer, for each asset security requirement, and each attack path endangering it, all the suitable mitigations, able to block at least one step of one of the aforementioned attack paths against the asset: this first result is saved in the Knowledge Base using the aforementioned concept of DPI. This preliminary task of the asset protection phase is detailed in Section 5.2.

Then, given all the possible DPIs, ESP infers all the possible combinations of DPIs, or *protection solutions*: each one must respect the fundamental requirement of being able to block all the attack paths against the application found in the risk assessment phase. Also, for a combination to be valid, protections deployed on the same asset must not present any incompatibility among them, as described in Section 2.1.3: ESP takes this problem into account, generating only valid solutions, given a set of incompatibilities reported by the software security experts, saved in the Knowledge Base using the relationships among protections defined in the meta-model for this purpose (see Section 3.3). Furthermore, since protections cause a decrease of application performances when deployed to protect its assets, in terms of CPU resources, memory occupation, and used network bandwidth for on-line protections. Thus, the global performance overhead introduced in the application by the inferred combinations of DPIs must be limited: the user can set an upper bound for the overheads, which will be respected by the inferred combinations of DPIs. As highlighted in Section 2.1.3, since it is unfeasible to actually measure the overheads by instrumenting the application, these are approximated using formulas specific for the protection

techniques, designed during the ASPIRE project in collaboration with protection authors: overheads depend also on the characteristics of the specific asset code protected with such techniques, thus these formulas are based on the complexity metrics, to take account of this dependence and obtain more reliable estimations. Evaluation of protection overheads is described in more detail in Section 5.4.3, while Section 5.3 reports on the generation by ESP of valid DPIs combinations.

However, the objective of ESP is presenting the best protection solution, so a way to compute the score of solutions must be devised. Indeed, multiple factors influence the effectiveness of a solution: the attacker profile, the possible attacks that must be deferred, the specific protection techniques used, the characteristics of assets code are all considered in assigning the score. The latter is assigned with a game-theoretic approach, which, taking into account all the aforementioned application, simulates a software protection "game": first, the defender protects the application and releases it to public, then the attacker tries the attack paths previously inferred in the risk mitigation phase, trying to breach at least one asset security requirement. A set of security measures, based on the assets software metrics, are used to model the resistance of assets against attack: the defender will first increase this measures through protections (extending the concept of transformation potency introduced by Collberg *et al.* [37]), then the attacker will try to decrease the measures until at least an asset is not defended anymore, so that its security requirements can be breached. The effectiveness of attacks in lowering the software measures is bound to the probability of success in executing the steps forming the path, which in turn depends on the attacker profile, and on the attacked asset metrics. This game-theoretic approach implements the simulation of the penetration testing executed by human security experts to test the protected application, thus fulfilling the related requirements for the risk monitoring phase (see Section 2.1.4).

Finally, the use of assets complexity metrics to assess protection efficacy have posed a cumbersome problem: as previously stated, the effectiveness of protections is closely related to the increase of such metrics in the protected code w.r.t. to the original one. Thus, to measure this increase, the metrics on the protected code must be calculated: for this, ESP relies on DIABLO, which evaluates metrics on the binary, after applying its protections. Thus, every time the score of a solution must be assessed, the protected binary should be built: however, the time needed to build the protected binary for each analyzed solution would not be feasible, especially when big applications, with not negligible compilation times, must be protected. Thus, a method is needed to predict the increase of complexity metrics of a specific asset, given the protections included in the solution that must be deployed on it: this problem has been solved, as presented in Section 5.4.2, with a set of predictors based on machine learning techniques: tests on real applications proved the high accuracy of the predictions obtained with this approach.

All the described algorithms, implementing the asset protection phase in ESP, present various configuration parameters that can be used to obtain increasing levels of result accuracy, at the expense of additional time needed by ESP to execute this phase: this permits the adaptation of this phase to the software life-cycle, as prescribed by the related requirement in Section 2.1.1.

2.2.4 Asset hiding phase

An additional mitigation phase has been introduced in ESP to hide assets from the attacker, mitigating the protections' side-effect of introducing fingerprints in the application, i.e., peculiar code structures or run-time behaviors that can indicate to the attacker the location of assets in the protected binary. This phase relies on the refinement of the best solution with additional protections, applied also on code areas that are not marked by the user as asset, so that the attacker can mistake such regular code areas as asset: this strategy is similar to the use of honeypots [116] to attract and identify intruders in computer networks, however, given the MATE scenario, aims to defer attackers, instead of identifying them. Three strategies to apply such

additional applications have been devised:

- fingerprint replication: additional fingerprints are added to the application, having a similar structure to the ones used during the assets protection phase, applying the same protections on non-sensitive code regions;
- fingerprint enlargement: the area of deployment of protections already present in the solution is extended to adjacent areas of code;
- fingerprint shadowing: existing fingerprints are concealed, applying other protections on the same asset.

The effectiveness of each strategy is related to the protection technique that introduced the fingerprint that must be concealed, and on the code complexity metrics of the protected application: this factors are combined in together in a *confusion index*, stating the additional effort needed by the attacker to locate the application assets, due to the additional protections inferred in this phase. Characteristics of the fingerprints, due to the protection techniques generating them, has been analyzed during the ASPIRE project, involving security experts and the developers of ASPIRE techniques in the definition of the fingerprints characteristics. Also overhead is taken into account in this phase (if not constrained, fingerprint enlargement and replication strategies could lead this phase to protect all the application code), using the evaluation formulas described in Section 5.4.3.

Thus, ESP must be able to refine the solution produced in the asset protection phase, adding to the existing solution additional protections to conceal their fingerprints: to choose the best protections, ESP must maximize the aforementioned confusion index while keeping the additional overheads below the user-defined limits. The solution adopted to find the best additional protections relies on the definition of a Mixed Integer-Linear Programming (MILP) optimization problem [19], based on the well known Knapsack Problem [47]. To solve the problem, ESP supports two external solvers: lp_solve, a FOSS solver, and IBM ILOG CPlex, a proprietary one. Also in this case, to satisfy the application lifecycle requirement stated in Section 2.1.1, the size of the problem can be limited to hasten its solution, paying the price of a lower accuracy in the found solution.

2.2.5 Complete workflow

All the phases described in this section can be organized in a unified software protection workflow, depicted in Figure 2.1, enabling ESP to safeguard an application to possible threats that can endanger the security requirements of the application asset, which are defined by the user. ESP, starting from the application source code, protects the code in an automated fashion, generating an application binary with all the protections needed to counter the possible threats deployed on it. Due this high grade of automation, ESP is usable also by a software developer that has no experience in the software security field. Conversely, ESP outputs also attack paths and analyzed mitigations, providing a software security expert with the information needed to manually assess the solution proposed by ESP, so that he or she can manually refine it, if deemed necessary. Thus, ESP satisfies the requirements related to interaction with the user, as defined in Section 2.1.1.

Summarizing, the workflow constitutes of the following phases:

1. *source code analysis*, a preliminary phase needed to generate automatically the model of the application source code structure, inferring the related information needed in the following workflows phases; the source code structure is formalized with classes and relationships of the meta-model defined in Chapter 3;

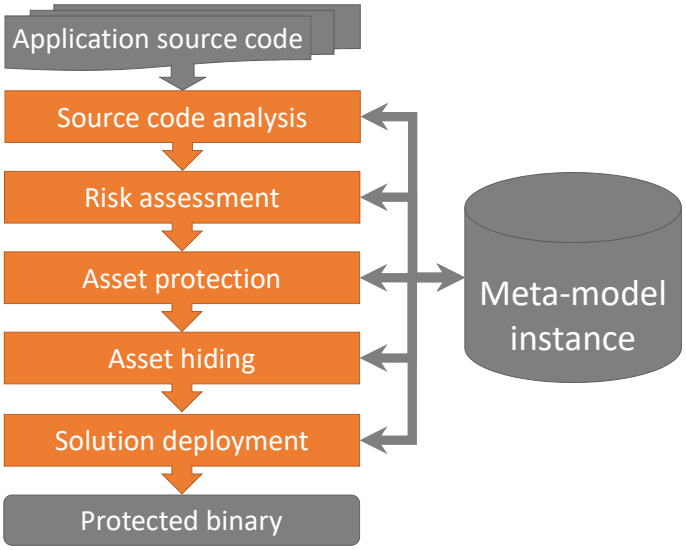


Figure 2.1: ESP workflow.

2. *risk assessment* phase, inferring possible attacks against application assets, with a reasoning process, based on backward programming, described in Chapter 4;

3. *asset protection* phase, selecting, among possible mitigations against the risk assessed in the previous phase, the ones that best perform in protecting the application, given its code structure and the attacker profile selected by the user; this phase is based on a game-theoretic approach detailed in Chapter 5;

4. *asset hiding* phase, refining the protection solution inferred in the previous phase with additional protections, applied also on areas of code not sensitive from a security point of view, with the objective of increasing the effort needed by an attacker to locate the assets in the protected application binary; this phase is based on the generation and solution of a customized Knapsack optimization problem, presented in Chapter 6;

5. *solution deployment*, driving the external software protection tools, presented in Section 1.1.1, to automatically generate an application binary protected with the best combination of protections obtained in the asset protection phase, and further refined in the asset hiding phase; details on how ESP drives the external protection tools are given in Appendix A.3.

2.3 Workflow execution example

This section provides an example of ESP execution on a simple application, in order to further ease comprehension of the automatic workflow phases. In particular, ESP has been used to protect Sumatra[7], a C console application used to compare DNA sequences. Sumatra is an open-source

[7] https://git.metabarcoding.org/obitools/sumatra/wikis/home/

Sumatra phase	Asset names	# assets
1	main	1
2	seq__readAllSeq2, cleanDB, addCounts, seq__fillHeader, seq__fillSeqOnlyATGC, seq__fillSeq, seq__fillDigitSeq, seq__readNextFromFilebyLine, uniqSeqsVector	9
3	compare1, calculateMaxAndMinLen, compare2, calculateMaxAndMinLenDB	4
4	seq__findSeqByAccId, seq__printSeqs printOnlySeqFromFastaSeqPtr, sortSeqsWithCounts, seq__getNext, reverseSortSeqsWithCounts, printHeaderAndSeqFromFastaSeqPtr, printOnlySeqFromChar, printResults, printOnlyHeaderFromFastaSeqPtr, printOnlyHeaderFromTable	11

Table 2.1: Functions marked as assets, grouped by Sumatra phases.

Class name	File	ApplicationPart	Asset	Datum	Code	Call	DatumItem
# instances	12	1159	25	909	225	708	1551

Table 2.2: Classes automatically instantiated in the source code analysis phase of ESP workflow, executed on the Sumatra application.

application. However, it can be treated as a commercial software, with a valuable comparison algorithm that must be protected against reverse engineering.

The DNA comparison can be subdivided into four consecutive phases:

1. the command line arguments are parsed; in function of their value, the appropriate functions are called to perform the DNA comparison;
2. the DNA sequences selected by the user are parsed and stored in memory, using a set of data structures for this purpose;
3. the latter are used by Sumatra core algorithm to perform the comparison of the sequences;
4. the comparison results are shown to the user.

Table 2.1 lists the 25 functions that have been manually marked as assets (using the code annotations described in Appendix A.3), subdivided by Sumatra phase. An example of asset is the function `compare1`, which contains one of Sumatra's core algorithms, designed to compare all the DNA sequences in a single dataset. For this example workflow, ESP has been instructed to find the protection solution best able to protect the confidentiality and integrity of all the aforementioned assets.

First of all, the application source code is automatically parsed, generating an instance of the application meta-model described in Section 3.2. The latter contains all the information needed in the subsequent phases of ESP workflow, including details on functions, global and local variables, and on the call graph. Table 2.2 lists the application meta-model instances automatically instantiated by ESP parsing the Sumatra source code.

Then, ESP risk assessment phase (detailed in Chapter 4) is executed, identifying the possible attack paths, ordered sequences of simple actions that a malicious actor may undertake in order to endanger the assets security requirements defined by the user. ESP has found 162 different attack paths against Sumatra's assets, combining a total of 150 different attack steps. For example,

an attacker may try to breach the integrity of the `compare1` asset (the function implementing the algorithm for the comparison of all the DNA sequences in one dataset) with the attack path constituted by the following steps:

1. the attacker executes Sumatra with a debugger attached, running a comparison between DNA sequences in a dataset;
2. the `compare1` function is consequently executed to perform the comparison, and the attacker can identify its position in the application binary thanks to the attached debugger;
3. knowing the asset's location in the binary, the attacker may tamper with it, again using the debugger, thus breaching the asset's integrity security requirement.

Knowing the possible attacks against Sumatra's assets, the asset protection phase (described in Chapter 5) can be performed. First, for each attack path previously found, ESP identifies the protections that can be applied to the endangered asset, in order to mitigate the analyzed path, thus generating the related Deployed Protection Instances (DPIs) that will be then combined into a comprehensive protection solution, able to protect all the assets against all the attack paths previously inferred. Each protection is associated with a level of mitigation against each kind of attack; for example, static remote attestation (see Section 1.1.3) has a high mitigation level against both static and dynamic tampering attacks, since this technique is designed to detect modifications of the monitored code. Then, ESP builds various combination of the DPIs, assessing the global effect of such protection solutions in deferring the possible attack paths found in the risk assessment phase, until all the possible solutions are generated or a termination condition is found (e.g., a hard time limit set by the user). The user can then select one of the solutions, deploy it as it is, or further refine it with the asset hiding phase (see Chapter 6) of the ESP workflow. In this way, additional protections, deployed on non-sensitive areas of code, are added to the solution, to further confuse a possible attacker.

Table 2.3 lists the DPIs constituting the best solution found in the asset protection phase by ESP on Sumatra. ESP has been instructed to consider only DPIs with at least a medium level of effectiveness against at least one attack path. For example, ESP has deployed three different protections to the `compare1` asset. Code mobility (see Section 1.1.3) moves to a remote trusted server a part of the asset's code, that is downloaded on the client machine only immediately before of its execution. In this way, the protection increases the effort needed by the attacker to reverse engineer the asset, thus preserving its confidentiality. However, the attacker could attach a debugger, manage to have the function downloaded and executed (e.g., using the attack path previously described) and analyze its behaviour at run-time, thus breaching its confidentiality. The asset is thus protected by this attack using the anti-debugging technique (see Section 1.1.4). Finally, both static and dynamic attempts to tamper with the asset's code are identified, monitoring it with the static remote attestation protection technique. This solution has been further refined with the asset hiding phase of the ESP workflow, adding 60 DPIs (on non-asset code areas) to the 27 DPIs constituting the original solution. For example, since the anti-debugging exhibits a peculiar fingerprint (the system call needed to attach the self-debugger on which the technique is based), the protection has been deployed also on non-sensitive code areas. In this way, an attacker looking for the anti-debugging protection fingerprint can be mislead to analyze one of these not valuable functions (from a security point of view).

2.4 Validation

This section elaborates on how ESP fulfils the requirements for the software risk management methodologies reported in Section 2.1. A more detailed evaluation is provided for each phase of

Protection name	Asset name	# mitigated attack paths
Anti-Debugging	cleanDB.r9	8
Code Mobility	printResults.r16	2
Source code obfuscation	printOnlySeqFromFastaSeqPtr.r20	8
Code Mobility	seq__readAllSeq2.r6	2
Anti-Debugging	calculateMaxAndMinLenDB.r13	6
Source code obfuscation	reverseSortSeqsWithCounts.r15	4
Anti-Debugging	seq__printSeqs.r8	2
Anti-Debugging	seq__fillSeqOnlyATGC.r4	4
Source code obfuscation	seq__readNextFromFilebyLine.r1	2
Anti-Debugging	addCounts.r10	2
Anti-Debugging	sortSeqsWithCounts.r14	4
Anti-Debugging	printOnlySeqFromChar.r21	2
Binary Obfuscation	main.r19	2
Anti-Debugging	seq__getNext.r0	4
Anti-Debugging	calculateMaxAndMinLen.r12	4
Code Mobility	compare2.r18	2
Static Remote Attestation	compare1.r17	3
Anti-Debugging	compare1.r17	6
Code Mobility	compare1.r17	2
Code Mobility	printHeaderAndSeqFromFastaSeqPtr.r24	2
Anti-Debugging	seq__fillDigitSeq.r5	2
Anti-Debugging	printOnlyHeaderFromTable.r23	2
Anti-Debugging	seq__fillSeq.r3	4
Anti-Debugging	seq__findSeqByAccId.r7	2
Code Mobility	seq__fillHeader.r2	2
Anti-Debugging	uniqSeqsVector.r11	6
Anti-Debugging	printOnlyHeaderFromFastaSeqPtr.r22	2

Table 2.3: Protection solution inferred by ESP for the Sumatra application.

the ESP workflow at the end of each chapter. Furthermore, this section presents an experimental assessment of ESP algorithms on three different applications, in order to prove its computational feasibility.

2.4.1 Qualitative evaluation

First of all, the workflow implemented by ESP models all the main tasks executed by software security experts tasked with the protection of a target application. The workflow comprises a phase to analyze the application structure, another to infer the possible attacks against the assets defined by the ESP user, and a phase to decide the mitigations most suitable to safeguard the assets security requirements. This workflow has been analyzed by software security experts involved in the ASPIRE project, and judged to correctly model the workflow they follow when tasked with the protection of an application.

Also, the implemented workflow is completely automated, with minimal interaction requested to the ESP user. In particular, the latter needs to provide to ESP the assets and the security requirements for them, which has been considered reasonable by the experts. Also, ESP can deploy the inferred protection solution automatically, since all the evaluated protections are implemented with the automatic protection tools described in Section 1.1.1. Thus, ESP can be used also by software developers with a limited background in software security. Also, the results of each phase in the ESP workflow are provided to the user in a human-readable format, e.g., sequences of simple attacker tasks for the risk assessment phase, ordered lists of protection techniques to be applied for each asset in the risk management phase. In this way, an expert using ESP can obtain automatically a protection solution for the application that he must protect, and can then evaluate it analyzing the results produced in each ESP workflow phase. If needed, he or she can then refine the automatically obtained solution.

All the decision processes implemented by ESP take into account the application structure, thus the produced results are tailored for the specific target application. All the concepts and

Application	C	H	Java	C++	**Total**
DemoPlayer	2,595	644	1,859	1,389	6,487
LicenseManager	53,065	6,748	819	-	58,283
OTP	284,319	44,152	7,892	2,694	338,103

Table 2.4: SLOC of ASPIRE use-case applications, used for the ESP validation.

relationships between them that are deemed useful in the software protection process have been formalized in a software security meta-model, presented in Chapter 3. Also, protection solutions are inferred taking into account the complexity metrics computed on the code comprising each asset, thus the efficiency of the solutions in safeguarding the assets reflects the real characteristics of the code.

During the ASPIRE project, three real-life Android applications, written in C, have been provided by the project industrial partners, in order to test the protection techniques developed during the project, and to validate the protection solution inferred by ESP on these applications. The use-case applications on which the ESP reasoning processes have been validated are a One-Time Password generator for home banking applications, an application licensing scheme, and a video player for Digital Rights Management (DRM) protected content. Details on these use cases have not been publicly released by their proprietors, and therefore are not included in this thesis. However, the Source Lines Of Code (SLOC) metrics for each use case application are reported in Table 2.4, proving the feasibility of analyzing and protecting complex applications with ESP.

Regarding the validation process, six experts from the three industrial partners have judged the results of the execution of ESP on the aforementioned use cases: in particular, two experts from each partner have validated the results on the use case built by their company. The data produced by ESP and validated by the experts has been the Attack Path (AP) inferred by the Risk Assessment Engine on the use cases assets, the protections deemed suitable by the Protection Enumerator to block these attacks, and the solutions produced by the Risk Mitigation Engine and Asset Hiding Engine combining the aforementioned protection; in particular, solutions have been validated in terms of achieved security for the assets, preservation of the application business logic, and containment of the inevitable slow-down of the protected application w.r.t. the original one. Furthermore, the AP have been compared with real attacks executed by each company's tiger team.

The validation has been positive: quoting from the related project deliverable [13], *"after the analysis of the validation data, the ASPIRE consortium concluded that the ADSS*[8] *has a very high potential even if it is not yet ready to be used to protect real applications"*. In particular, the combinations of protections inferred by the Risk Mitigation Engine and the Asset Hiding Engine have been judged as correct, i.e., they can be applied to the use case source codes, obtaining protected binaries that are unaltered in terms of business logic and are still usable, without an excessive overhead introduced by the protection. Also, the proposed solutions have been deemed effective to block the AP inferred by the Risk Assessment Engine, and also many real attacks executed by the professional tiger teams of the project industrial partners. The main flaw of ESP reported by the experts is that APs produced by the Risk Assessment Engine are too coarse-grained, since the attack rules are generic: future work on ESP will have to address this limitation, expanding the existing attack rules, e.g., by modelling actual attacks against real

[8] ASPIRE Decision Support System, the name originally adopted for ESP in the context of the ASPIRE project.

application listed in databases such as the CVE[9] and CWE[10], or in automated attack frameworks such as Metasploit[11]. The complete report on the ESP validation is presented in the related project deliverable [13].

2.4.2 Experimental assessment

ESP has been tested on three applications written in the C language, with increasing size and complexity. Statistics about such test cases are reported in Table 2.5. The ESP complete workflow has been executed on each application, running the asset protection (see Chapter 5) and hiding (see Chapter 6) phases multiple times, varying the number of different PIs available to protect the applications' assets. The tests have been executed on an Intel i7-8750H 2.20 GHz with 32 GiB of RAM, using Java 1.8.0_212 under GNU/Linux Debian 4.18.0. Figure 2.2 depicts the obtained results, showing the total ESP computation time, along with the time needed for the risk assessment, asset protection and hiding phases.

In all the executed tests, the time needed to analyze the applications source code and to generate the application meta-model instance has been negligible (less than 1s) and is thus not drawn in Figure 2.2. Also the time needed to deploy the solution is excluded from the aforementioned figure, since it is completely dependent on the time needed for the external protection tools execution, and is therefore not relevant for the assessment of ESP computational feasibility. The time needed to execute the risk assessment phase is clearly not dependent on the numbers of PIs available to protect the application, since the algorithm evaluates possible attacks on the vanilla application. However, Figure 2.3 highlights that time needed for such phase is strongly dependent on the application code complexity (e.g., SLOC, number of assets and functions). The asset protection phase is by far the most computationally intensive, especially with high numbers of available PIs. The order used to deploy different protections to the same asset may have a deep impact on the achieved level of security (see Section 2.1.3). Thus, in the asset protection phase, ESP must not only assess the different combinations of PI that can be applied to each asset, but also the possible permutations of each combination of PI, causing the strong correlation of asset protection algorithms complexity with the number of available PIs. The same holds also for the asset hiding phase, even if less time needed to execute the latter, compared to the asset protection phase.

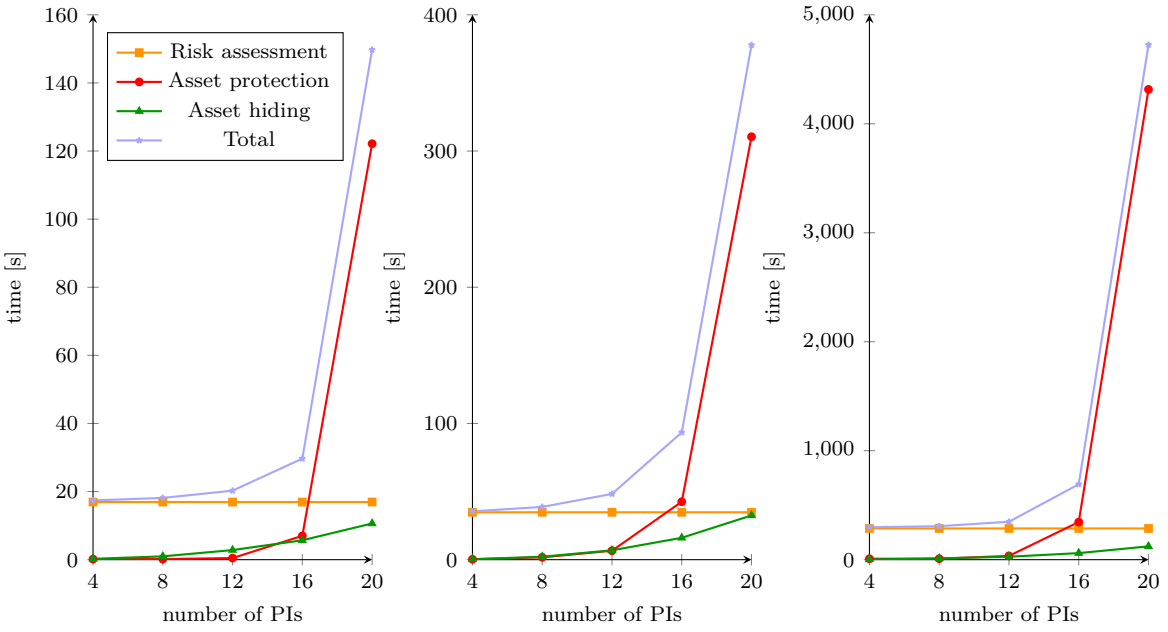
Application	SLOC	Functions	Assets		
			Code	Variables	Total
A	443	18	2	2	4
B	1029	47	12	3	15
C	3749	178	26	13	39

Table 2.5: Code statistics of applications used for ESP experimental assessment.

[9] https://cve.mitre.org/

[10] https://cwe.mitre.org/

[11] https://www.metasploit.com/



(a) Application A (b) Application B (c) Application C

Figure 2.2: ESP execution times on applications reported in Table 2.5, with increasing number of available PIs.

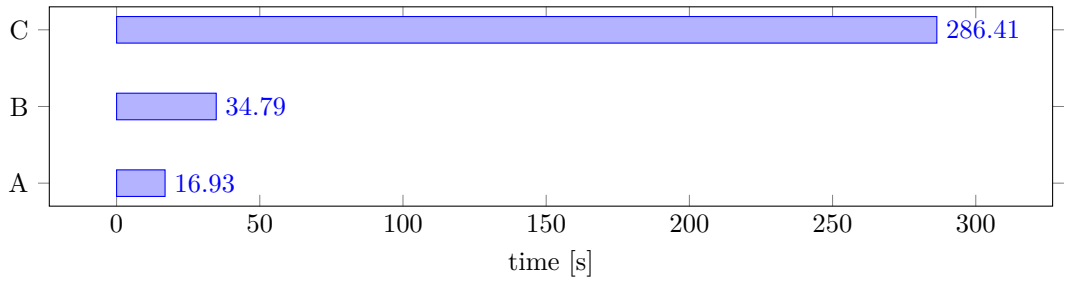


Figure 2.3: Execution times for ESP risk assessment phase on applications reported in Table 2.5.

Chapter 3

Software security meta-model

> Information is the resolution of uncertainty.
>
> Claude Shannon

This chapter, which is a reworked excerpt of the paper *"A meta-model for software protections and reverse engineering attacks"* [15], presents the software security meta-model on which the Knowledge Base is modelled, introducing how the a-priori knowledge gathered among software security experts during the ASPIRE project, and all the relevant information about the target application code structure is formalized, in order to be used by the ESP reasoning processes.

Information formalized by the meta-model can be distinguished in three main categories:

- *generic a-priori knowledge*: all the information about obtained from software security experts, e.g., generic attacks, protection techniques, security requirements;
- *target a-priori knowledge*: the description of the target application, i.e., concepts and classes that can be automatically inferred by ESP analyzing the target source code (e.g., functions, variables, CFG and call graph), or manually specified by the user (e.g., assets and their security requirements);
- *a-posteriori knowledge*: the results of the ESP reasoning processes, such as the attacks against the target application assets inferred in the risk assessment phase, and combinations of protections devised in the risk mitigation phase to safeguard application assets from the aforementioned attacks.

The meta-model is implemented by the Knowledge Base as an OWL-2 ontology. At the start of the ESP workflow, this ontology contains the aforementioned generic a-priori knowledge. Then, is enriched with target a-priori and a-posteriori knowledge during the various phases of the ESP workflow, described in Section 2.2. The Knowledge Base, to manipulate the ontology file, leverages methods from an external library, providing an API for OWL-2 ontology management[1].

For the sake of readability, the meta-model is split in four smaller ones. This section is organized accordingly in the following sections:

- Section 3.1 describes the *core meta-model*, presenting the basic classes and concepts used among all the ESP workflow phases;

[1] https://github.com/daniele-canavese/ontologies

- Section 3.2 details the *application meta-model*, formalizing the main concepts used to describe the target application source code structure, built at the start of the ESP workflow parsing the target application source code, and used by the all the ESP reasoning processes in the subsequent phases of the workflow;
- Section 3.3 outlines the *protection meta-model*, with the classes and relationships representing general information about protections techniques, and the concept of applying them to specific assets of the target application, in order to model the decision processes of the asset protection (see Chapter 5) and hiding (see Chapter 6) phases of the ESP workflow;
- Section 3.4 describes the *attack meta-model*, containing the classes used to model general information about attacks in a MATE scenario, and actual attacks that can be mounted against the target application assets, inferred in the risk assessment phase (see Chapter 4);
- Section 3.5 reports on the ESP requirements satisfied with the definition of the software security meta-model, and its implementation as the ESP Knowledge Base.

3.1 Core meta-model

The core meta-model formalizes the main concepts and relationships between them involved in all the phases of the software protection workflow implemented by ESP. Figure 3.1 sketches the UML class diagram of the core meta-model, which comprises the classes needed to model the target application, the user-defined assets and security requirements, the attacker, the potential attacks targeting assets security requirements inferred in the risk assessment phase (see Chapter 4), and the software protections supported by ESP and used in the asset protection and hiding phases (described respectively in Chapters 5 and 6).

The main class is Application, abstracting the applications or libraries that must be protected. An Application is composed by ApplicationPart instances, representing functions, code regions (syntactically valid sections of functions, see Section 6.3.1 for more details), and variables, both global and local. An Asset is an instance of the ApplicationPart class associated with a set of security requirements, (as reported in Section 2.1.2, ESP supports confidentiality, integrity, and execution correctness) targeted by an attacker and that must be safeguarded with some protection. All the Asset instances must then have at least one hasRequirement association with the SecurityRequirement enumeration, containing all the security requirements the ESP user may assign to an asset.

The AttackTarget class models a possible target of an attacker, aiming at breaking the asset security requirements. The meta-model associates each AttackTarget instance with one and only one Asset, using the threatens association, and with one and only one SecurityRequirement element, via the affects relationships. If the user defines multiple secuirty requirements for an asset, several AttackTarget on the same asset are instantiated, since an attacker may target each of the asset security requirements.

As it will be shown in Chapter 4, attacks are modelled in the ESP risk assessment phase as ordered sequences of steps. For example, an attacker targeting the integrity of a function in the application, e.g., a license check, can decompile or disassemble the application binary, identify the asset (i.e., the license check function), and then tamper with it, thus breaking its integrity security requirement. Such basic attacker tasks are modelled with the class AttackStep. Instances of this class may have one or more hasTarget relationships with instances of the AttackTarget class. Attack steps that model some preparatory actions, needed by the attacker to execute the following attack steps (e.g., attaching a debugger to the application, before inspecting the execution flow in a targeted function), are not associated with any target.

Attacks are modelled with the AttackPath class, whose instances are ordered sequences of attack steps. It should be noted that not only the last attack step in a path can breach an asset security

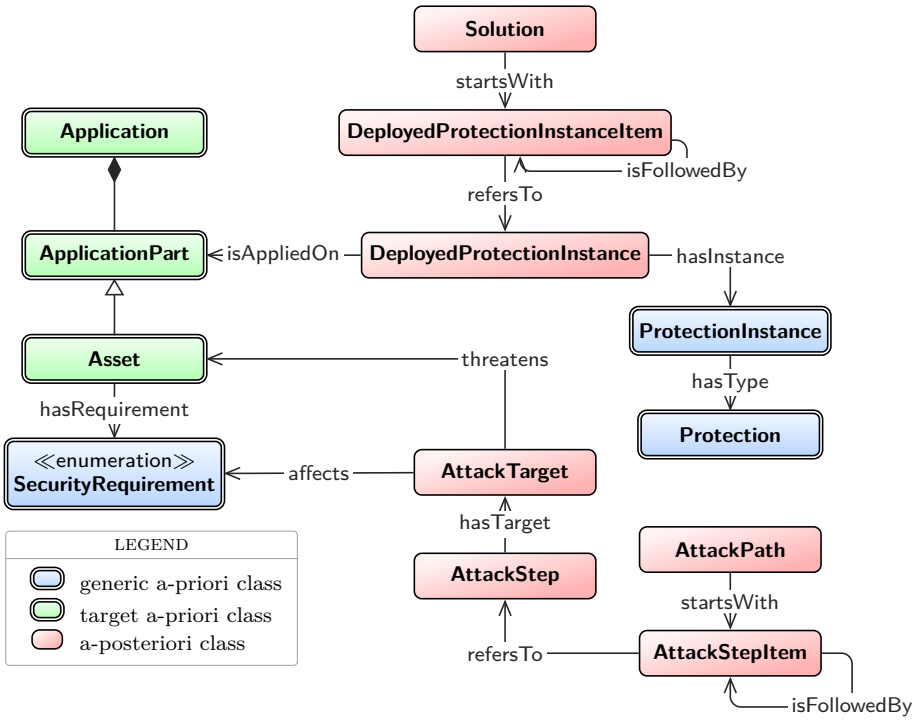


Figure 3.1: UML class diagram of the core meta-model.

requirement, e.g., an attack step threatening the confidentiality of an asset may lead to another step that breaches the asset integrity. The attack step ordering is formally enforced with the class AttackStepItem, whose instances are associated with a single AttackStep object using the refersTo association, and with the next step in the attack path via the isFollowedBy association. Each AttackPath instance is related to one AttackStepItem instance, representing the initial step in the path, using the startsWith relationship.

Generic protections techniques are modeled with the Protection class. A protection enforced with a specific tool, with a specific configuration, is represented as an instance of the ProtectionInstance class. Every ProtectionInstance object has a hasType association that binds a ProtectionInstance object with its generic protection (i.e., a Protection instance), and a isEnforcedWith association with one or more ProtectionTool instances, modeling all the tools supported by ESP to actually deploy the protection. For example, the control flow flattening obfuscation technique (see Section 1.1.2) is modeled as a Protection class instance.

Various PIs must be deployed to the application assets, in order to slow down an attacker. Thus, the DeployedProtectionInstance class is introduced in the model, representing a PI deployed to a generic application part. This association is not directed to the not the asset concept, but to the more generic ApplicationPart class, since ESP protects also non-assets in the asset hiding phase (see Chapter 6), in order to confuse and consequently slow down the attacker. Instances of the DeployedProtectionInstance class are related via the hasInstance and isAppliedOn associations to respectively a ProtectionInstance and ApplicationPart instances, representing the PI and the application part where the former is deployed.

Sets of DPIs, representing the solutions inferred by ESP in the asset protection (see Chapter 5) and hiding (see Chapter 6) phases of its workflow, are modeled with the Solution class. To find the best trade-off between the level of security achieved and the introduced overhead, different

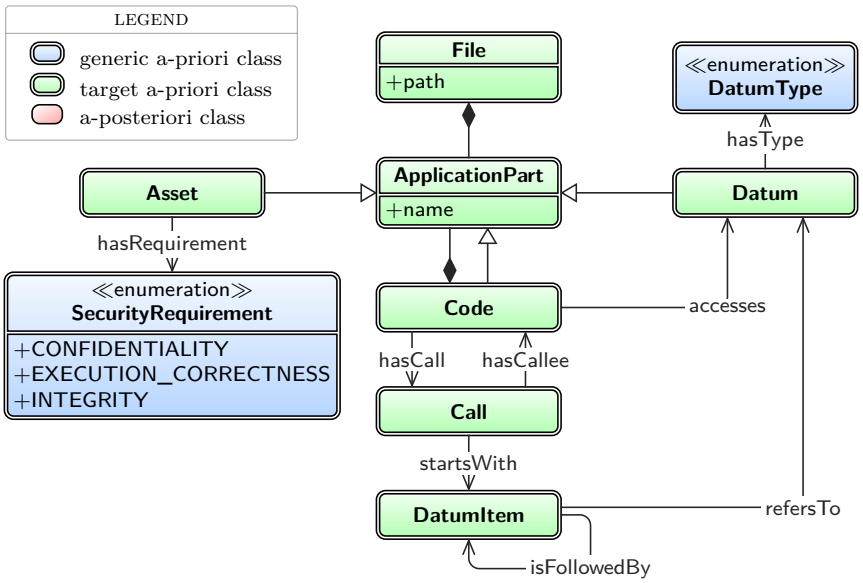


Figure 3.2: UML class diagram of the application meta-model.

solutions are devised by such phases, thus the meta-model will contain multiple Solution instances for the same application.

When more than one protection is deployed to the same asset, the order of application is important, since it could lead to different levels of obtained security and even to incoherent cases (see Section 5.3). Thus, an ordering between the DPIs in a solution is enforced using the DeployedProtectionInstanceItem class, representing a DPI in a solution. Every DeployedProtectionInstanceItem object is related to a DeployedProtectionInstance object with the refersTo association. Each Solution instance will have an association startsWith with an DeployedProtectionInstanceItem instance, in order to individuate the first DPI. The DPI ordering in the solution is then modeled with the isFollowedBy relationships between the DeployedProtectionInstanceItem instances.

3.2 Application meta-model

The meta-model sketched in Figure 3.2 contains the fundamental information about the application used by all the phases of the ESP workflow. This data is needed to protect its assets, in order to preserve the security requirements of the latter from the attacks mounted by the attacker.

The ApplicationPart class models the various components of an application. Each application part has a name attribute and it is contained into a source file modelled with a homonym class, indicating its location in a file system with the path element. All the ApplicationPart instances can be assets, code or data, modelled by three distinct sub-classes.

The Datum sub-class is used to represent a generic variable or function parameter. Each datum is defined by its type (e.g., string literal, integer variable), represented by the DatumType class and hasType association. This information is important for ESP, since data protections supported by ESP are only applicable to specific data types, e.g., the literals obfuscation described in Section 1.1.2 can be applied only to string literals and integer constants.

The Code sub-class is used to represent functions or any code region[2], i.e., a container of other application parts (e.g., a function contains variables, but also other smaller code snippets). This can be represented with the containment relationship between the ApplicationPart and Code classes. A code region can also access a (local or global) variable, which is represented with the accesses association. In addition, since also the call graph of the application may prove useful, especially when inferring attacks in the ESP risk assessment phase, each function call is represented with an instance of the Call class. The caller code is related to the call via the hasCall 1-to-1 association, while the call is bound to the callee with the hasCallee 1-to-1 association. Each call instance comprises also to the parameters passed to the called function, organized in an ordered list. A parameter is thus represented through the DatumItem class, associated to the correspondent Datum instance with the refersTo relationships and the next item with the isFollowedBy association. If the called function has at least one parameter, the Call instance will comprise a startsWith association with a DatumItem instance representing the first call parameter. When it is relevant to take into account multiple calling sites to the same callee in a caller function, this can be done by considering multiple ApplicationParts in the function, and by relating each of them to the callee with hasCallee.

As already described in the core meta-model, assets are represented as instances of the Asset class, which in turn is a sub-class of ApplicationPart, and are associated with their security requirement with the relationship hasRequirement to elements of the SecurityRequirement enumeration. ESP supports as requirements confidentiality, integrity, and execution correctness, which are detailed in the introduction of Chapter 4. However, the meta-model does not restrict the usage of these requirements, but allows the security expert to add additional ones if needed.

3.3 Protection meta-model

The protection meta-model, outlined in Figure 3.3, comprises the classes and relationships related to the protections, used in the asset protection (see Chapter 5) and hiding (see Chapter 6) workflow phases. This data can be used to protect the security requirements of the assets against attacks. This meta-model is able to represent not only the protection relationships, but can be also used to precisely describe how an application must be protected.

The Protection class is related with SecurityRequirements values by means of the enforces association. This relationship models the abilities and purposes of applying a given protection. Also, the Protection class has various association loops useful to model protection synergies and forbidden precedences. The shouldBePrecededBy and shouldNotBePrecededBy associations are respectively used to define that a DPI should or should not be preceded by another DPI of a given kind. This is useful when choosing the best solution since one protection can make another, previously applied protection stronger (e.g., static remote attestation, presented in Section 1.1.3, can be made more resistant to attacks if coupled with anti-debugging, detailed in Section 1.1.4), but applying one protection can also make a later one weaker (e.g., control flow obfuscations, presented in Section 1.1.2, if applied first can negatively impact the data flow analysis that checks preconditions for applying data obfuscations detailed in Section 1.1.2), thus affecting the aggressiveness with which the data obfuscation can be applied. Furthermore, the cannotBePrecededBy relationship is used to model impossible sequences of protections that can lead to incoherent or non-compilable applications (e.g., software remote attestation is usually the last protection to be put, since altering the code after its deployment will trigger an invalid attestation).

The ProtectionTool class comprises all the tools supported by ESP that can be used to deploy a protection on an asset or application part. The supported PIs are related to their tool via the isEnforcedWith association.

[2]A formal definition of code region is presented in Section 6.3.1.

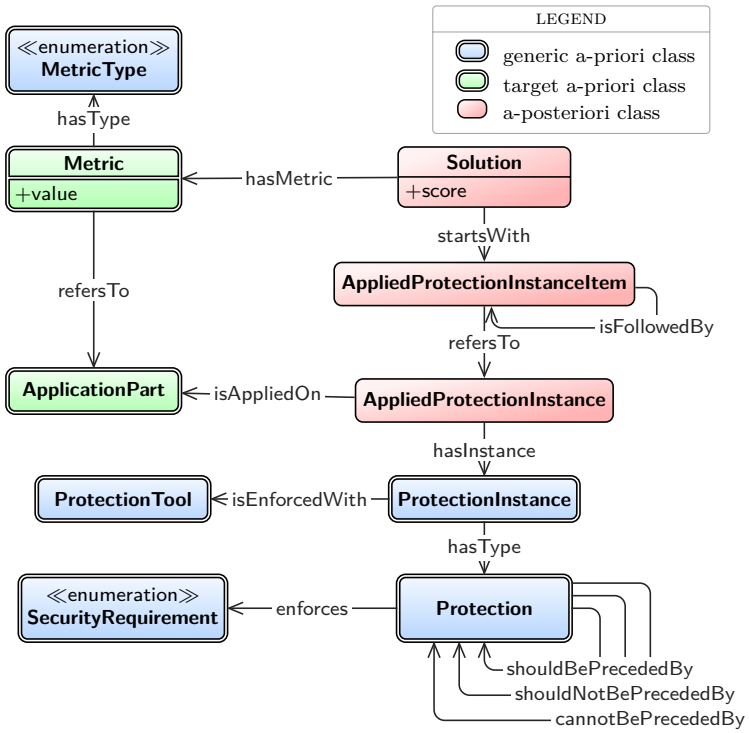


Figure 3.3: UML class diagram of the protection meta-model.

Concluding, the Metric class instances model the value of complexity metrics computed over an application part (see Section 5.4). The value attribute represents the quantitative value of the metric, while the kind is represented via the hasType association towards an enumeration MetricType containing all the available metric categories (see Section 5.4.1). The refersTo and the hasMetric associations direct towards respectively the relative application part and the current protection solution. Complexity metrics are used in the asset protection and hiding phases of the ESP workflow to adapt their decision to the actual characteristics of the protected code; in particular, for the asset protection phase, the increase of complexity metrics evaluated on a protected asset w.r.t. the original asset code is used as an index of employed protection effectiveness, following the concept of transformation potency proposed by Collberg *et al.* [37]. Based on this indexes, in the aforementioned ESP workflow phase a score (see Section 5.5.1) is assigned for each inferred protection solution. To model this, the Solution contains the score attribute.

3.4 Attack meta-model

The attack meta-model, depicted in Figure 3.4, comprises all the classes and relationships employed by ESP in the risk mitigation phase (see Chapter 4). These allow to represent the attacker, his attacks and their effects on the application and the protections.

Attackers are represented with the Attacker class, related with the hasExpertise association to the AttackerExpertise enumeration, representing the four levels of attacker expertise supported by ESP, defined in Section 2.2.2. Also, the solution is related to a specific attacker via the hasAttacker relationship to explicitly indicate that it was generated in function of the attacking profile specified by the ESP user.

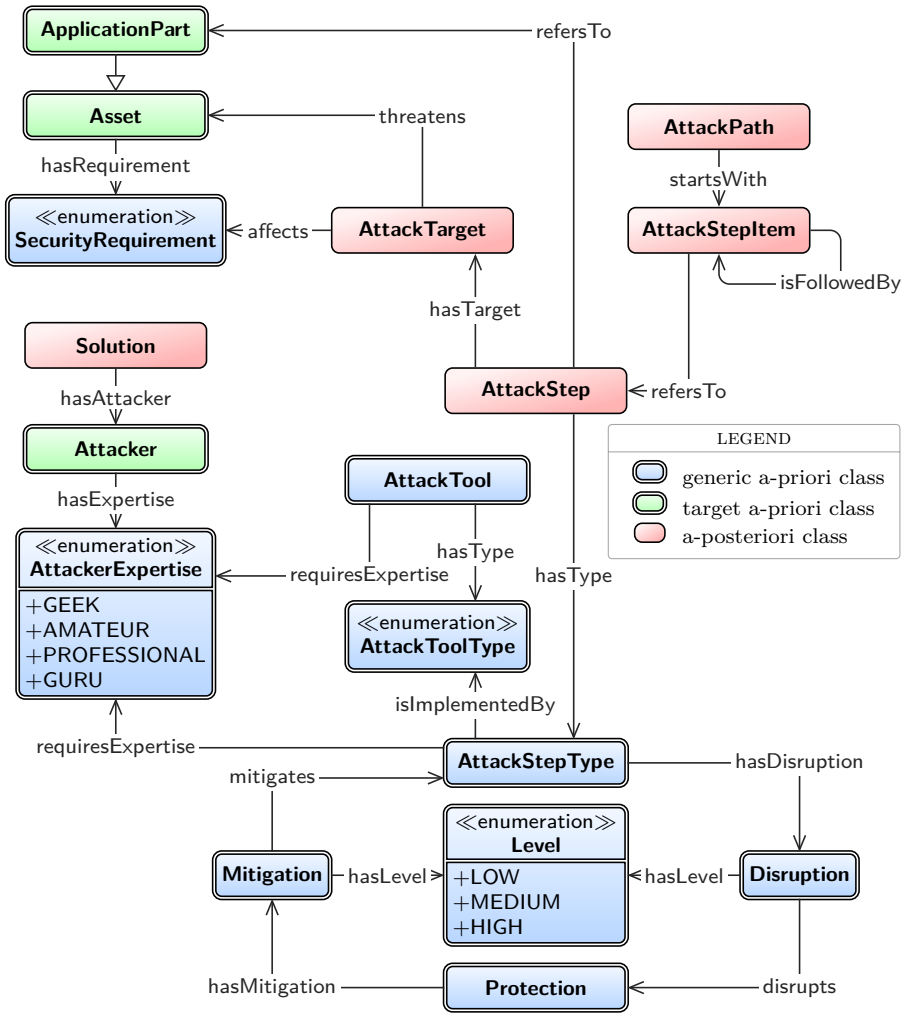


Figure 3.4: UML class diagram of the attack meta-model.

Attack steps are related to the targeted application part. This is represented through the refersTo association. An attack step can threaten a security requirement of an asset, a fact modelled with the AttackTarget class and its relationships. For example, if a variable 'a' is an asset whose confidentiality must be preserved, the attack step 'locate the variable a in the function b' is related to the function 'b', and has a relationship with the attack target for the confidentiality of the asset 'a'.

An AttackStep may need a minimum level of attacker expertise to be mounted, thus representing its base difficulty level. The requiresExpertise relationship models this concept, used in the risk mitigation phase of ESP to evaluate the probability of an attacker successfully executing an attack path (see Section 4.4). Similarly, the meta-model comprises the requiresExpertise relationship between AttackTool and AttackerExpertise instances, which permits the classification of tools, in function of the minimum level of skills the attacker should have to be capable of employing it.

All attack steps are associated with a specific type, e.g., dynamic analysis or static tampering. This is modeled through the AttackStepType class and the hasType association. An attack step does not need to represent a complete attack, but it can be also used to model a preliminary step such

as 'execute the application'. Also, the requiresExpertise association represent that an attack step type needs a minimum level of expertise to be executed by an attacker.

An attack step type (e.g., a debugging attack) can be executed by one or more different attack tool types (e.g., a debugger). This is modeled via the isImplementedBy association with the AttackToolType enumeration. The latter is associated with the AttackTool class, with the hasType relationship, comprising the known attack tools (e.g., IDA Pro).

The hasMitigation relationship models the fact that a protection can mitigate an attack step type. For example, this is able to express that the opaque predicates obfuscation technique (see Section 1.1.2) can be employed to increase the effort needed to perform both static and dynamic analysis attacks. The Mitigation class models the protection mitigation, and is related with the attack step type it is able to temper through the mitigates association. It also permits to specify a non-numeric level of effectiveness by using the hasLevel association and the Level enumeration. Conversely, an attack can be mounted with the objective of removing a protection, partially or completely. This concept is used in the assessment of the effectiveness of the solutions inferred by ESP in the asset protection phase of its workflow (see Section 5.5), and is represented the hasDisruption relationship with one or more Disruption class instances. Analogously to the mitigation case, this class specifies the protection that is affected by an attack via the disrupts association and the effectiveness level of the disruption with the hasLevel relationship.

3.5 Meta-model validation

This section presents the validation of the meta-model presented in this section w.r.t. the requirements for ESP defined in the Section 2.1.

Indeed, the meta-model is able to perform its main purpose, i.e., the modelling of all the data needed to perform the decision processes executed throughout the ESP workflow, along with their results. All the general a-priori classes have been instanced in the ESP Knowledge Base with background knowledge gathered among the software security experts involved in the ASPIRE project, which validated during the project results validation phase the ability of ESP to model their generic software security knowledge on which they base their decisions.

The application meta-model (see Section 3.2) contains all the information needed to characterize the application structure in detail, and to model the security requirements provided by the ESP user for the application assets; via an automated parsing of the target program source code (see Appendix A.2), ESP is able to infer all the target a-priori classes of the meta-model. For the risk mitigation phase (see Chapter 4), the attack meta-model described in Section 3.4 is able to model the attack paths as ordered sequences of attack steps, and comprises all the information needed to represent the attacker profile, its skills and the attack tools that he or she may be able to use. For the asset protection and hiding phases (see Chapters 5 and 6), the protection meta-model provides all the needed classes to model the protections as ordered sequences of specific protection instances. The latter are associated with the tools (see Section 1.1.1) that automates their deployment, in order to obtain the protected binary without requiring intervention from the ESP user.

However, the applicability of the meta-model is not limited to the ESP Knowledge Base. Indeed, concepts from various taxonomies and surveys of reverse engineering and software protection techniques, previously presented in literature, can be easily mapped to the classes and associations of the meta-model. Ceccato *et al.* [31] developed a taxonomy of concepts used by attackers to describe their attack methods and the reasoning processes behind their choices. The authors built such taxonomy analyzing penetration tests and public challenge reports produced by both amateur and professional hackers. Along with the taxonomy, the authors presented also four models able to express the causal, conditional, temporal and instrumental relationships between various attacker activities, such as high-level comprehensions tasks, attack tool selection

and creation activities, and choices made to removing or circumvent protections. The meta-model is able to map all the top-level concepts in the aforementioned work. Examples are assets, attack tools, attack steps, all of which are present both in the meta-model and in Ceccato *et al.* models and taxonomies. The full mapping has been reported in [15]. The meta-model also completely covers the taxonomy detailed in Collberg *et al.* seminal work [37], organizing various obfuscation techniques against reverse engineering (see Section 1.1.2). The latter are easily supported in the meta-model through the Protection, ProtectionInstance and other classes and associations of the protection meta-model (see Section 3.3). Furthermore, the authors of the aforementioned work proposed to evaluate the potency of obfuscation techniques using software complexity metrics, which are expressed in the meta-model with the related class. Another interesting taxonomy from Schrittwieser *et al.* [108] models attack techniques against software obfuscation, based on code analyses. Such techniques are partitioned in categories based on the attack goal, the abstract technique leveraged to reach such goal (e.g., "locating code through static analysis") and whether or not the attack technique can be fully automated. All the combinations considered by the authors can be modelled with the attack meta-model, in particular via multiple instances of the AttackStepType and AttackToolType. Finally, it is interesting to compare the meta-model with the taxonomy of software integrity protection by Ahmadvand *et al.* [4]. The latter defines generic attacks, that can be easily mapped via instances of the AttackStepType class of the meta-model, which is more precise, being able to represent complex attack paths tailored to endanger security requirements of specific assets. Also, the taxonomy lacks the concept of deployed protections and solutions, since it has been designed to describe and categorize integrity protections. The taxonomy includes information about the life cycle of such protections, describing the management and production stages of the targeted application. Indeed, this would be an interesting addition to the meta-model, extending its applicability to situations when applications must be protected without access to the source code. The taxonomy includes also the high-level concept of overhead, which is covered by the meta-model and ESP using software metrics (see Section 5.4.3).

To best of the author's knowledge, this is the first meta-model targeting software security. However, various meta-models have been defined for computer network security. Various meta-models and modelling languages have been proposed to represent threats in enterprise networks. Sommestad *et al.* [114] defined the Cyber Security Modeling Language (CySeMoL), which is able to model computer systems in an enterprise network scenario. The authors have also presented a method to infer threats directed to such systems via an inference engine based on the models characterized with CySeMoL; success probability of the inferred attacks is also evaluated. An extension of this work, presented by Valja *et al.* [121], includes an improved security analysis, which handles attacks from malicious actors external to the enterprise network, and also by legitimate users inside the analyzed network.

A meta-model to assess the security of cloud applications has been presented by Kritikos *et al.* [80], with the definition of the Cloud Application Modelling & Execution Language (CAMEL), a domain-specific language able to describe the design and the security requirements of cloud applications. It also allows the validation of the model against a set of constraints.

Mouelhiv *et al.* [92] presented a meta-model to model access control policies, focusing on mutation analysis, a testing technique for security policies based on the voluntary injection in the analyzed policies of flaws (mutation), to assess the efficiency of security tests; the meta-model is able to represent the mutation operators used in such tests.

Chapter 4

Risk assessment

> To know your Enemy, you must become your Enemy.
>
> Sun Tzu

This chapter presents the risk assessment method used by ESP to infer possible attacks against the application assets in the vanilla application, i.e., the unprotected program. As detailed in Section 2.2.1, ESP, prior to this phase, executes a preliminary analysis of the source code, enriching the Knowledge Base with the description of the application structure, instancing the related meta-model classes described in Section 3.2. The risk assessment phase employs this information, in order to infer attacks tailored to the specific application examined. Each attack aims to breach a security requirement of an application asset; these are provided by the user, annotating the application source code as detailed in Section A.3.

ESP supports three security requirements for assets: *confidentiality*, *integrity*, *execution correctness*. A confidential asset must not be understandable by an attacker: this is the case for example of an algorithm that must remain secret, in order to safeguard the software company IP, or of a data structure that holds personal information of the user, such as the credit card number in an e-commerce application. The integrity requirement specifies that an asset must not be modifiable by an attacker: for example, in the case of a soccer video-game, the variables holding the characteristics of the soccer players, or the match score, can be marked with the integrity requirement, in order to avoid an attacker to cheat. The execution correctness requirement is a stronger form of integrity for code: the latter must not be modified, but must also be called as expected by other functions. For example, an attacker that wants to remove a license check function, marked with this requirement, may modify the function in order to not perform the check, but can also avoid the function to be executed, removing any calls by other functions to the license check one.

ESP models attacks against such security requirements with *attack paths*, which are sequences of simple tasks that an attacker may execute (e.g., locating a variable in the binary, debugging a function at run-time), called *attack steps*: this modelling approach has already been used by other works in literature for risk assessment of computer networks [52, 56]. Each attack step is associated with a set of postconditions, which model the result of a successful execution by the attacker of the simple task related to the step, and a set of preconditions, which must be fulfilled by the results of other steps preceding it in the path. Attack steps may contain also preconditions about the application structure: for example, an attack step modelling the debugging of a function, can be executed only if the analyzed asset is a code and not a variable.

ESP, to infer the possible attack paths against the asset security requirements of the target

application, relies on an external inference engine, where attack steps are modelled as logical rules. At the start of the risk assessment phase, ESP instructs the engine about the structure of the application, modelling each concept of the application meta-model (see Section 3.2) as an axiom, i.e., a fact that engine knows to be true. Then, ESP queries the engine, for each asset security requirement, to prove that the attacker can breach it: to do this, the engine tries to find every possible chain of rules (i.e., a sequence of attack steps forming an attack path), which can be ultimately proven with a subset of the aforementioned axioms about the application structure. The engine carries out this task using a backward programming algorithm (see Section 1.2.2). Other works in literature [97, 73] have already used a similar approach to assess risk for computer networks. To ease readability, in the remainder of this chapter attack steps and axioms are expressed with the mathematical proofs notation.

The attack steps are modelled with the following inference rules:

$$\frac{P}{C}\ id,$$

Where:

- id is the name that identifies the attack step;
- P is a set of facts, called *premises*: if these facts are verified, the attack step can be executed by the attacker;
- C is a set of facts, called *conclusions*: these facts will be verified if the attack step can be carried out by the attacker.

The user can influence the time needed to infer the possible attack steps with two parameters. First, he or she can set a hard time limit (expressed in seconds). Also, he or she can decide a limit for the backward programming algorithm, in term of number of traversed relationships among application parts (e.g., the maximum distance among application parts in the call graph), when the algorithm is searching for an attack step that, with its conclusions, can satisfy a premise of another attack step.

Finally, the attack paths are saved in the ESP Knowledge Base, structured using the related classes described in the meta-model (Section 3.4). The attacks are then used in the risk mitigation phase of ESP: during the latter, the protections best suitable to protect the application assets, against the attack paths previously found by the inference engine.

This chapter is a reworked version of the publication "Towards Automatic Risk Analysis and Mitigation of Software Applications" [104]. It is organized in the following sections:

- Section 4.1 describes the preliminary of the application structure modeling in the inference engine, previously obtained by ESP parsing the source code;
- Section 4.2 defines the rules needed to model the attacker goals;
- Section 4.3 details the methodology for modelling attack steps, and how the inference engine combines them in attack paths;
- Section 4.4 reports on the method used to assess the risk of the inferred attack paths;
- Section 4.5 contains a comparison of the obtained results w.r.t. the ESP requisites defined in 2.1.

4.1 Application structure modeling

This section presents the modelling of all the preliminary information needed by the inference engine to find possible attacks against the application, comprising assets with their security requirements and the application structure. All this information is automatically retrieved by ESP in the preliminary source code analysis phase of its workflow.

First, the existence of an application a is defined with the following fact:

$$\overline{application(a)}$$

Then, the application structure must be formally modelled. Each basic component of the code can be either a code or a datum (a constant or a variable); for example, if the application a contains a function f and a global variable d, this can be modelled with the following facts, using by abuse of notation the symbol $\triangleright$ to indicate the containment relationship:

$$\overline{code(f)} \quad \overline{datum(d)} \quad \overline{f \triangleright a} \quad \overline{d \triangleright a}$$

The containment relationship permits to organize the application structure in a hierarchical way. For example, if the function c contains a local variable v, and a code snippet s, we can define the function structure with the following facts:

$$\overline{datum(v)} \quad \overline{\triangleright f} \quad \overline{code(s)} \quad \overline{\triangleright f}$$

Then, a code c accessing a variable v can be expressed with the following fact:

$$\overline{accesses(c, v)}$$

Also the call graph of the application can be modelled; if a function f_1 containing a call to a function f_2, the following fact holds:

$$\overline{calls(f_1, f_2)}$$

Finally, the assets and their security requirements must be defined. If the ESP user requests the confidentiality security requirement $\overline{\mathcal{C}}$ for a function f_1, integrity $\overline{\mathcal{I}}$ for a function f_2 , and execution correctness $\overline{\mathcal{E}}$ for a function f_3, the following facts are asserted:

$$\overline{code(f_1)} \quad \overline{code(f_2)} \quad \overline{code(f_3)} \quad \overline{\mathcal{C}(f_1)} \quad \overline{\mathcal{I}(f_2)} \quad \overline{\mathcal{E}(f_3)}$$

4.2 Attacker goals modeling

After modelling the application structure, and the assets with their security requirement, the basic rules that define the goals of the attacker can be defined. ESP, for each asset security requirement, will query the inference engine to find a set of rules that lead to breaching the requirement, denoted formally as the logical negation of the requirement. For example, if the application contains a variable v whose confidentiality must be safeguarded, the inference engine will try to find all the possible sequences of attack steps that lead to the following outcome, which is a goal G of the attacker:

$$\neg\, \mathcal{C}(v) \iff G. \tag{4.1}$$

Security requirements are characterized by the different ways in which the attacker can breach them. The confidentiality of an asset a_1 is breached if the attacker finds a way to retrieve its content (instructions if a_1 is a code, or data if a_1 is a variable):

$$\frac{contentRetrieved(a_1)}{\neg\, \mathcal{C}(a_1)}$$

The integrity of an asset a_2 is breached if the attacker manages to change it, modelled with the following rules:

$$\frac{changed(y)}{\neg \mathcal{I}(a_2)}$$

To model the breaching of execution correctness, two rules are needed. Formally, the attacker breaches the execution correctness of a code, either by modifying its instructions, or by avoiding the execution of the code, removing at least one call to the function. Thus, if a function f is marked with the execution correctness requirement, the latter can be modelled with the following rules:

$$\frac{changed(f)}{\neg \mathcal{E}(f)} \quad \frac{calls(g, f) \quad changed(g)}{\neg \mathcal{E}(f)} \tag{4.2}$$

With the definition of the following facts in the inference engine, ESP informs the latter of the application structure and of the security requirements of the assets. Thus, the attacks able reach the attacker goals, i.e., to breach the security requirements of the assets, will adapt to the application structure.

4.3 Attack steps and paths modeling

This section elaborates on the formal modelling of attack steps, basic and indivisible tasks that an attacker may carry out, and that can be combined into complete attack paths against the application assets.

First, a set of basic attack steps must be defined. First, the attacker may retrieve the content of a hard-coded asset h, e.g., a constant or a set of instructions, by locating it in the binary, and inspecting the latter manually:

$$\frac{hardcoded(h)}{contentRetrieved(h)} \; statLocate(h)$$

Conversely, the attacker may retrieve the instructions constituting a code asset c at run-time, for example by means of step-by-step execution with a debugger, provided that the execution flow of the application can lead to the target code. Similarly, an attacker can read the value of a local variable v locating it in memory at run-time. This leads to the following attack steps:

$$\frac{code(c)}{contentRetrieved(c)} \; dynLocate(c)$$

$$\frac{datum(v) \quad v \rhd f}{contentRetrieved(v)} \; dynLocate(v)$$

If an attacker wants to locate a datum d to retrieve its value, he may try to first identify and execute at run-time a function f that uses the aforementioned datum, thus finding it in memory when the instruction of the function f that accesses the datum d is executed:

$$\frac{code(c) \quad contentRetrieved(c) \quad accesses(c, v)}{contentRetrieved(d)} \; dynLocate(d)$$

Also, if an attacker wants to locate a function f, and he or she has already retrieved a function g that calls f, then by simply following the call, looking at the target address of the call instruction, he or she can retrieve the function f in the binary:

$$\frac{calls(g, f) \quad contentRetrieved(g)}{contentRetrieved(f)} \; followCall(g, f)$$

Thus, if such datum d is a confidential asset, one of the possible attack paths, leading to the attacker goal G of breaching d confidentiality can be obtained by chaining the previous attack steps:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{code(c)}}{hardcoded(c)}}{contentRetrieved(c)}\ statLocate(c) \quad \dfrac{}{accesses(c,v)}}{contentRetrieved(d)}\ dynLocate(d)}{\dfrac{\neg\mathcal{C}(d)}{G}}$$

In this example, the attacker first identifies the address of the function c in the code section of the application binary, analyzing statically the latter. Indeed the $statLocate(c)$ attack step can be executed, because c, being a code, is hard-coded. Then he or she attaches a debugger to the application and sets a breakpoint at the address of the function c previously located. Then, since the code c accesses the variable v, by executing c step-by-step the attacker is able to find the variable v when c accesses it, and thus, knowing v's location in memory, can retrieve its content. Therefore, the confidentiality of the asset d is breached, and the attacker goal G is proved. Clearly, instead of this hybrid approach (i.e., by resorting to both static and dynamic analysis), the attacker can breach d confidentiality in a completely dynamic manner, finding the code c observing the execution flow of the application. Similarly, the call graph of the application can be used to define other attack steps. For example, an attacker may locate a function f_1 at run-time, if he has previously located a function f_2 that calls f_1, and the function is executed.

For tampering attacks two types of attack steps may be modelled. As for the retrieval of content, tampering may be carried out statically only on hard-coded assets, while it can be executed dynamically on every kind of asset. Given a code or datum x, the attacker may change it statically or dynamically with the following attack steps, which require the attacker to know the location of x:

$$\dfrac{contentRetrieved(x) \quad hardcoded(x)}{changed(x)}\ statChange(x)$$

$$\dfrac{contentRetrieved(x)}{changed(x)}\ dynChange(x)$$

Thus, attack paths that endanger the integrity and execution correctness security requirements of an asset can be obtained by chaining the steps required to locate the asset that must be tampered, and the steps in which the attacker actually tamper with the asset. Given a function f whose execution correctness must be preserved, i.e., f must not be tampered with, and all the calls to f must be executed, this attacker goal G can be obtained with the following two example attack paths:

$$\dfrac{\dfrac{\overline{calls(g,f)} \quad \dfrac{\dfrac{\overline{hardcoded(g)}}{contentRetrieved(g)}\ statLocate(g)}{changed(g)}\ statChange(g)}{\neg\mathcal{E}(f)}}{G}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{calls(g,f)}\quad\dfrac{\dfrac{}{hardCoded(g)}}{contentRetrieved(g)}\ statLocate(g)}{contentRetrieved(f)}\ followCall(g,f)}{changed(f)}\ dynChange(f)}{\neg\,\mathcal{E}(f)}}{G}$$

In the first attack path, the attacker first locates the function g. He can do it statically since g is hard-coded. Then he or her changes it, removing from g the call to the asset f. In this way, the execution correctness of f is broken. In the second attack path, the attacker locates statically the function g as before, but in this case uses its address to set a breakpoint in a debugger attached to the application. For example, this can be useful in the case of an indirect call, when the target address of the call (i.e., the address of f) is not known a priori, since it is evaluated by the program at run-time. Thus, he will execute the application, which will stop at the breakpoint set at the start of function g. With an execution step-by-step of the function g, the attacker finds the call to function f, follows it an then retrieves the address of the address function, thus being able to change the contained instructions and breach the execution correctness of the asset f.

Thus, starting from apparently naive attack steps, the inference engine can combine them in rather complex attack paths. All the concept and relationships among them, modelling the application structure, are obtained automatically by ESP, thus these attack paths can be obtained in a completely automated fashion, with the users asked only for the security requirements of the assets.

Other attack steps have been devised to model attacks on applications that include network connections in their execution flow, modelling sniffing, spoofing and remote code injection attacks. However, these attack steps cannot be employed in a completely automated workflow by ESP, since it would require the identification of the specific functions that perform such network activity. This is complicated to do automatically, especially if the application uses custom network libraries. Since the requirements for ESP (see Section 2.1) state that all workflow phases must be executed automatically, such network-based attack steps are not deemed relevant for the scope of this thesis. Still, use of the spoofing attack step to model attacks against an One-Time Password (OTP) generator has been reported in a conference publication [16].

The chaining of the rules is executed by the inference engine via a custom backward programming algorithm, called by ESP for each asset security requirement. Given this goal, the algorithm starts by negating the asset security requirement and then finds all the possible attack steps that can lead to the asset security requirement negation: each parallel attack step leads to a separate attack path. Then, it goes on building all the possible paths: for each of the latter, the algorithm consider completely inferred a path when the last attack step requirements can be proved only with the axioms detailing the application. Therefore, ESP builds for each asset security requirement an attack graph, which contains four different attack paths to breach the execution correctness of a code c_2, which is called by another function c_1.

4.4 Risk probability

In this section, a method to assess the probability of the threats against the application assets is presented. ESP, when presenting the attack paths to the user, includes also such a probability: this is useful for experts that want to manually analyze the protections inferred by ESP in the risk mitigation phase, since they can identify the assets that are most at risk, and consequently focus on the protections targeting the most threatened assets.

Instead, it will not be used to drive the risk mitigation phase (see Section 5.5.1): the motivation

behind this is that the probabilities inferred in this phase refer to the unprotected application, while the risk mitigation phase uses a set of formulas tailored to assess the effect of attacks on protected assets, to assess the effectiveness of the inferred mitigations.

The risk probability is based on the types of attacks inferred, and on the presumed skills of the attacker, given the profile chosen by the user at the start of the ESP workflow. Given all the inferred attack paths AP_i, all their attack steps $AS_{i,j}$, and all the assets a_k, the *risk* $\Omega^{\epsilon}_{\mathrm{AP}_i}$ of an attack path AP_i executed by an attacker with expertise ϵ is defined as:

$$\Omega^{\epsilon}_{\mathrm{AP}_i} = \pi(\mathrm{AP}_i, \epsilon)\, \Gamma_{\mathrm{AP}_i}$$

where Γ_{AP_i} represents the damage resulting from a successful attack path, and $\pi(\mathrm{AP}_i, \epsilon)$ is the probability of an attacker being able to successfully carry out the attack path, given its expertise ϵ. To do so, it must be able to execute all the attack steps needed to complete it. Thus, the probability of executing an attack path is evaluated taking into account the probabilities of all the attack steps constituting the path:

$$\pi(\mathrm{AP}_i, \epsilon) = f\left(\pi(\mathrm{AS}_{i,1}, \epsilon), \pi(\mathrm{AS}_{i,2}, \epsilon), \dots\right).$$

The ESP user can decide the function f that must be used to combine the probabilities of the attack steps belonging to the same path, choosing among a worst case analysis, with $f = min$, a best case analysis, with $f = max$, and an approximation of the probabilities composition, with $f = \cdot$.

Γ_{AP_i} is a quantitative measure of the damage resulting from a successful attack path. It is evaluated as the sum of the damage $\Gamma_{\mathrm{AS}_{i,j}}$ from each attack step:

$$\Gamma_{\mathrm{AP}_i} = \sum_j \Gamma_{\mathrm{AS}_{i,j}} = \sum_j \sum_k (W_{a_k} b(a_k, \mathrm{AS}_{i,j}))$$

where W_{a_k} is a user-defined asset weight, and $b(a_k, \mathrm{AS}_j)$ is a function, evaluated by ESP, that returns the fraction of the security properties of the asset a_k that are breached by the attack step AS_j, that is, it returns 1 if all the security properties are compromised and 0 if none[1].

The attack step probabilities $\pi(\mathrm{AS}_{i,j})$, for each combination of attack step type, and level of attacker skill, have been obtained during the ASPIRE project, interviewing the software security experts involved in the project during ESP design phase.

4.5 Validation

In this section, the methodologies presented in this chapter, used by ESP to infer possible attack paths against the application assets, are compared w.r.t. the ESP requisites listed in Section 2.1.

Regarding the requisites on the usage scenario of ESP, detailed in Sec. 2.1.1, they are satisfied by the risk assessment phase. First, the profile of the attacker is taken into account, when the risk of each attack path is evaluated. Second, the execution of the backward programming algorithm used by the inference engine can be constrained, using a hard time limit, and also setting a maximum length for the attack paths: a deeper and more time-consuming search is suitable prior to the distribution of the application, while, when patches are released and time is an issue, a shallow search for possible new attack paths derived from the introduction of the patch code

[1] Thus, the whole asset weight assigned by the user is gained by the attacker when all the security properties are compromised.

can be performed, limiting the time needed using the aforementioned constraints. Third, the results of the mitigation phase, i.e., attack paths, are easily readable, being modelled as ordered sequences of simple attack steps: furthermore, logical inferences that do not correspond to real tasks executed by the attacker are not included in the paths, to avoid confusion for the user. Furthermore, a risk level for each inferred attack path is presented to the ESP user, which can easily identify the most problematic threats to the application.

Analyzing the requisites for the risk assessment phase, described in Sec. 2.1.2, they are mostly satisfied. First, ESP supports not only the security requirements defined by the NIST for risk monitoring [68], i.e., confidentiality and integrity, but also a stronger form of the latter, called execution correctness, and applicable only to functions: this security requirement prescribes that the function marked with it must not only be preserved from modification, but it must be also called as originally devised by the application developer (e.g., a license check that must not be circumvented). Furthermore, ESP is able to build attack graphs for every asset in the application, due to the definition of logical inference rules that define attack steps, i.e., simple attacker tasks. The chaining of these steps in attack paths is subject to the application structure, which is automatically modelled with logical facts in the inference engine: thus, the relative requirement is satisfied. Finally, a probability of execution for each attack path is evaluated, based on the attacker skills, and the probability of execution of each attack step: the latter are based on information gathered among the software security experts involved in the ASPIRE project. The complexity metrics of the software are not taken into account in the attack path probability evaluation: however, they are taken into account later in the ESP workflow, during the risk mitigation phase, when the effectiveness of the inferred protections in protecting the assets, against the attack path inferred in the risk assessment phase, is tested with a game-theoretic approach based on this metrics.

Finally, in the validation phase of ESP at the end of the ASPIRE project [13], software security experts analyzed also the attack paths inferred in the risk mitigation phase. In particular, the ones inferred by ESP on the ASPIRE use cases (see Section 2.4) were compared with actual attacks executed by highly skilled white-hat hackers on the same test applications. In general, ESP attack paths covered the real methodologies of the aforementioned hackers. However, they were deemed as too general to model with a sufficient amount of detail such attacks. Indeed, new rules can be added to the inference engine, in order to better model real attacks on software: however, as in the case of network-related attacks, the main problem is inferring automatically the semantics of the code that enables the execution of such complex attack step. Thus, this is still an open problem, which the author is actively investigating, in particular focusing on automated binary exploitation frameworks [112, 66], which could be used to actually assess the vulnerabilities of the applications in an automated manner. Their results could be still modelled as attack paths, thus being compatible with the remainder of the workflow.

Chapter 5

Asset protection

> Chess is a war over the board. The object is to crush the opponent's mind.
>
> Bobby Fisher

This chapter presents how ESP is able to assess the effectiveness of mitigations against the risks on application assets. ESP combines different protection techniques, such as the ones presented in Section 1.1, in a comprehensive *protection solution*, able to safeguard a given application against possible attacks that can endanger its assets. This task is typically carried out manually, by software protection experts, that analyzes the application source code, reason on the possible attacks that can be carried out against the application assets, and decide the most suitable protection techniques to block the attacks. This decision is mainly based on their knowledge and past experience in the software protection field.

Indeed, this is not an easy task, even for an expert. First, there are various possible security requirements for an asset, and there does not exist one protection able to preserve all of them: for example, code obfuscation techniques (Section 1.1.2) are able to preserve the confidentiality of a protected function or code region[1], but are not capable of blocking attacker attempts to tamper with them, which is the scope of anti-tampering protections (Section 1.1.3). Also, the applicability of a protection is limited to one type of asset: different techniques are necessary to protect code areas or the values of variables and constants. Attackers have a broad range of tools at their disposal, such as debuggers, automatic code analysis tools, disassemblers and decompilers, leading to a plethora of different types of attacks, and, again, one protection cannot block all of them: thus, a simple mapping between protections and enforced security requirements is clearly not enough, and a set of protections must be applied to each asset in order to block all the possible attacks endangering its requirements. Another problem is assessing the real effect of a protection technique, when applied on a specific function or variable, in increasing the difficulty of executing possible attacks against them: again, a simple mapping between protections and blocked attacks is not sufficient, since the effectiveness of protections may vary in function of the structure of the code on which they are applied. For example, the effectiveness of CFF is directly related to the number of basic blocks in the CFG of the protected code, since each basic block can be put in a different case of the CFF loop, leading to a wide CFG difficult to analyze for an attacker.

[1] A code region is a section of a function that is syntactically valid if parsed in isolation; a formal definition of code regions is provided in Section 6.3.1.

Furthermore, combining different protections techniques is a process that must be carried out with extreme care, and even the order in which the protections are applied to the same asset can have a beneficial or detrimental effect on the obtained security. For example, if CFF is applied on a function that has been previously protected with opaque predicates, the latter become more difficult to be removed, since the two branches introduced by each opaque predicate will be put in different cases of the CFF loop: on one hand, this increases the parallel cases of the flattened CFG, hardening its analysis, while on the other hand identifying that the two branches pertain to the same opaque predicates takes more effort for the attacker, since this correlation is not evident anymore in the CFG; however, this effect is not obtained if the opaque predicates are inserted on a code already flattened, since each of them will be inserted in one of the CFF cases, without having their branch split. A case of combination that ultimately leads to an application not working at all is the deployment of any kind of obfuscation after remote attestation (Section 1.1.3): since the latter monitors attested code in order to find any attempts of tampering with it, the modifications introduced by an obfuscation technique will be mistaken for a tampering attack, triggering a reaction, for example the abrupt termination of the application. Therefore, all the possible interactions among used techniques must be taken into account when protecting an asset.

When protections techniques are combined, also the relationships among assets must be taken into account. A simple example can be a whole function whose integrity must be preserved, which contains a sub-routine that should remain confidential: while theoretically these two assets can be seen as separated, having different security requirements, this containment relationship must be taken into account to avoid incompatible combinations of protections; taking again the example above, if the whole function integrity is protected with remote attestation, and afterwards the confidentiality of the contained code region is ensured with some kind of obfuscation, the remote attestation checks will be triggered and the application will not work.

Finally, it should also be noted that protections usually take a toll on application performances: an example is the anti-debugging technique (Section 1.1.4), which prevents an attacker from launching a debugger, by actually debugging the protected areas of code with a custom debugger out of the attacker control. However, a context switch is necessary every time the execution flow enters a protected code, thus slowing the application. Also, on-line techniques (Section 1.1.3) need to exchange data over the Internet, in order to communicate with their server counterpart: if this communication is executed across unreliable networks, such as the one used by mobile phones, this may lead to a slow-down of the application, for example if the application is protected with code mobility (Section 1.1.3) and the code needed to continue the execution is not downloaded on time; also, if the user of the mobile application has a limited data plan for his phone, this may lead to undesirable charges by the mobile ISP. Thus, when choosing the protections, the overhead introduced by them in the application must be taken into account, with the protected application that must be thoroughly tested after deployment of protections: this may lead to various time-consuming cycles of protection and testing, until the wanted trade-off between security and usability of the application is found.

Summarizing, when protecting an application, an holistic approach is necessary: structure of the application and of the assets that must be protected, security requirements that must be preserved, possible attacks that can endanger them, the pros and cons of the available protection techniques, and the interactions among them must all be taken into account, in order to obtain an application that is secure, trying to leave user experience untouched. With this approach in mind, a set of algorithms have been devised to automate this cumbersome process, with the rather ambitious goal of modelling the mental processes of an expert in the act of protecting an application. The automated workflow may be useful for software security experts in protecting applications, presenting them with a protection solution they can further refine instead of starting from scratch, but also for software developers, allowing them to protect their applications without

being versed in software security.

Using a *divide and conquer* approach, this complex problem has been subdivided into two smaller ones, each solved by a different ESP component:

- the identification of the protections that are able to protect an asset security requirement, being capable of blocking at least one attack endangering the requirement; this problem is solved by the Protection Enumerator;
- the inferral of protection solutions for the target application, able to safeguard all the security requirements of all the assets comprised in the application, leaving the business logic of the application untouched, and avoiding to introduce too much overhead in the application execution; the algorithms automating this process are comprised in the Risk Mitigation Engine.

This reasoning processes are a fundamental part of the ESP workflow, enabling it to infer the a-posteriori classes of the protection meta-model detailed in Section 3.3, and ultimately leading to the individuation of the most suitable combination of protections that, if deployed on the program source code with the Solution Deployer, is able to safeguard the user-defined security requirements of a target application assets. To correctly infer the suitable combination for a specific target application, the Protection Enumerator and Risk Mitigation Engine need a Knowledge Base containing the following information:

- generic software protection knowledge gathered from software protection experts, i.e., the a-priori information in the software protection meta-model presented in Chapter 3;
- a list of assets and security requirements, manually specified by the target application developer, either by annotating the code as presented in Section A.3, or using the User Interface;
- a user-defined set of maximum application performance overheads (see Section 5.4.3) that a solution can cause after being deployed on the target application with the Solution Deployer;
- structure and software metrics of the unprotected application source code, obtained respectively from the automatic analysis executed by the Source Code Analyzer via CDT (Appendix A) and the Metrics Framework via the ACTC (Section 5.4), i.e., a completely inferred application meta-model (Section 3.2);
- APs discovered by the Risk Assessment Engine against user-defined assets security requirements, i.e., a completely inferred attack meta-model (Section 3.4);
- optionally, a set of user-defined constraints to avoid the evaluation of all possible solutions by the Risk Mitigation Engine: the user can specify a time limit, or a maximum number of evaluated solutions.

This chapter is organized as follows:

- in Section 5.1, the ESP workflow that leads to the identification of the best protection solution for a given application is presented;
- Section 5.2 elaborates on how the Protection Enumerator builds the DPIs used by the following steps of the workflow, identifying, for each asset security requirement, ESP supported APIs that are suitable to safeguard the aforementioned requirements;
- in Section 5.3, the Risk Mitigation Engine *solution walker* is presented, i.e., the algorithm used by the Risk Mitigation Engine to compute all the feasible combinations of DPIs previously built by the Protection Enumerator;

- Section 5.4 lists the quantitative software metrics evaluated on the target application that are taken into account by ESP, with a brief description of the algorithms used to produce them;
- Section 5.5 details the Risk Mitigation Engine *solution solver*, i.e., the algorithm used by the Risk Mitigation Engine to find, among the feasible solutions identified by the Risk Mitigation Engine solution walker, the one that can best safeguard the application assets against the attacks discovered by the Risk Assessment Engine (see Chapter 4);
- Section 5.6 validates the asset protection decision processes w.r.t. the ESP requisites defined in 2.1.

5.1 Protection decision workflow

This section presents the workflow followed by ESP to infer appropriate protection solutions for a target application: each solution is associated with a score, which indicates the effectiveness in slowing an attacker executing the APs previously built by the Risk Assessment Engine. Among the found solutions, the best ones will be shown to the ESP user[2], so that he or she can select one of them to be deployed on the application code via the Solution Deployer, which will produce a binary protected with the DPIs comprised in the chosen solution.

Figure 5.1 depicts the aforementioned workflow, highlighting the involved ESP components, along with calls among them, the meta-model objects they use, and the meta-model classes inferred in the various steps of the workflow (indicated with a circled red number), which are the following:

1. ESP calls the Protection Enumerator which, for each specific asset security requirement (also called Protection Objective (PO)) previously defined by the user, enumerates the PIs that can defer at least one of the APs that endanger the PO, thus building and saving in the Knowledge Base a set of DPIs for each PO;
2. ESP calls the Risk Mitigation Engine: in the following workflow steps, its two sub-components, the solution walker and the solution solver, will find the protection solutions, i.e., combinations of DPIs found in the previous steps, which will be shown to the user in the last workflow step; this step is further detailed in Section 5.2;
3. the Risk Mitigation Engine calls its first sub-component, the solution solver;
4. the solution solver calls the solution walker, which reads from the Knowledge Base the previously built DPI set, and generates a solution (i.e., a subset of DPIs from the aforementioned set, organized in an ordered sequence of DPIs) that is both correct and effective, i.e., if deployed on the target application, produces a protected binary that behaves correctly during execution, and protects all the POs in the application; this reasoning process is presented in Section 5.3; then, the walker returns the solution to the solver;
5. each time the walker generates a valid solution, the solver calculates the score of such solution with a game-theoretic approach detailed in Section 5.5, using the metrics predicted by the Metrics Framework and the APs inferred by the Risk Assessment Engine; then, if all the possible solutions have been evaluated, or if at least one of the user-defined conditions for Risk Mitigation Engine termination has been met (e.g., maximum execution time or number

[2]The number of solutions shown to the ESP user can be set by him or her in the User Interface.

of evaluated solutions), the solution solver terminates, otherwise another solution is asked to the walker (i.e., the workflow returns to step 4);

6. having added the inferred solutions to the Knowledge Base, the Risk Mitigation Engine can halt its execution, returning control to ESP;

7. ESP gathers the inferred solutions from the Knowledge Base, and presents them to the user.

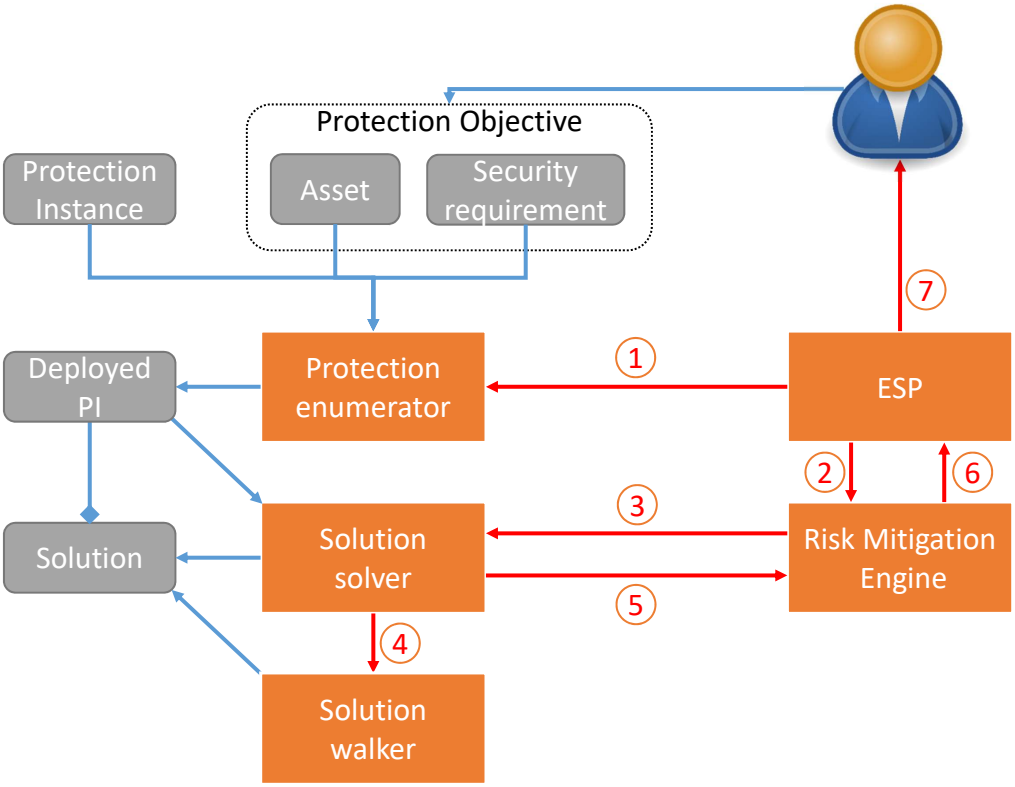


Figure 5.1: ESP protection decision process, with main software protection meta-model classes and ESP components involved in the process.

When presented with the inferred solutions, the user can select one of them and can alternatively:

- deploy the selected solution on the target application, obtaining the protected binary: in this case, ESP will call the Solution Deployer, described in Appendix A.3;
- further refine the chosen solution with additional protections on non-asset APs, in order to increase the effort needed by the attacker to locate the protected assets; with this choice, ESP will give the selected solution in input to the Asset Hiding Engine, presented in Chapter 6.

5.2 Deployed protection instances enumeration

This section details the internal processes of the Protection Enumerator, corresponding to step 2 of the protection decision workflow (Section 5.1). The main task of this component is instancing

in the Knowledge Base the DPIs that can safeguard one or more security requirements of an asset in the target application. In general, given a PO, i.e., the security requirement of a specific asset, the Protection Enumerator infers that deploying the considered PI on the aforementioned asset can be useful if (as specified by the expert a-priori knowledge in the Knowledge Base) the protection implemented by the PI is in general able to safeguard the PO security requirement: in this case, a new DPI is instanced in the Knowledge Base, related to the asset and PI taken into account.

First of all, the Protection Enumerator needs to enumerate the PIs available in the system running ESP. As detailed in Section 3.3, a PI represents a specific protection technique implementation, stating the tool that must be used to deploy the technique, along with its configuration parameters: for example, the remote attestation technique may be deployed by ESP using the ASPIRE implementation of such technique (Section 1.1.3), specifying the algorithm that must be used to compute the hashes of attested regions (e.g., Blake, SHA1). In particular, the Protection Enumerator parses a set of eXtensible Mark-up Language (XML) files, contained in the Knowledge Base source code folder (see Appendix A), specifying for each PI:

1. the PI name;
2. the name in the Knowledge Base of the implemented protection;
3. the name in the Knowledge Base of the protection tool that must be used by the Solution Deployer to deploy the protection (e.g., Tigress or ACTC);
4. the protection tool parameters values characterizing the PI; for protections deployed by the ACTC, this is represented by the related ASPIRE annotation (see Section A.3), while for Tigress techniques implementation the options that must be included in the command line[3];
5. the formulas to compute the introduced overhead if the PI is deployed on a code region, depending on the metrics of the latter (see Section 5.4.3).

The parsed PIs are therefore saved in the Knowledge Base: the user can disable one or more PIs, using the User Interface. Note that PIs are not hard-coded in the PIs as a-priori information, in order to ease the addition of other PIs by the user, which simply needs to add a new XML file (or edit an existing one) adding the requested information for the new PI. For example, this is useful to drive DIABLO, since it has a parameter that specifies the percentage of basic blocks to which the binary code obfuscations (Section 1.1.2) must be applied: a higher value for this parameter means a more thorough obfuscation, at the expense of an increased overhead. The default XML file contains a set of PIs for each binary code obfuscation, with different values for this parameter: however, a user can edit the related XML file easily to specify its set of parameter values.

Subsequently, the Protection Enumerator can infer the DPIs that are suitable to protect an application, and that must be considered by the Risk Mitigation Engine when finding the possible protection solutions; for each PO, a DPI, representing the deployment of the PI on the asset constituting the PO, is added to the Knowledge Base if:

- from the a-priori expert knowledge in the Knowledge Base, the protection technique implemented by the PI enforces the PO security requirement;

[3] As specified in the section *"Options"* for each transformation page in the Tigress website; for example, http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html#options lists the command-line options for the flattening transformation.

- the PI is applicable to the considered asset type; for example, Tigress variable obfuscation (Section 1.1.2) is applicable only to integer variables;
- specific requirements of the considered PI are fulfilled; for example, the ASPIRE implementation of code mobility (Section 1.1.3) can be applied exclusively to whole functions, so, if an asset is only a region of a function, the Protection Enumerator builds a DPI related to the code region representing the whole function comprising the asset.

Also, the user can set the following constraints to limit the number of DPIs inferred by the Protection Enumerator, such as a minimum Level that a Mitigation of a Protection must have to be deemed useful in deferring an AP. Finally, the DPIs fulfilling the aforementioned requirements and user-defined constraints are added to Knowledge Base, and will be considered by the Risk Mitigation Engine for inclusion in the produced protection solutions, as reported in the following sections.

5.3 Inference of valid solutions

This section details the solution walker algorithm, part of the Risk Mitigation Engine, tasked with steps 4 and 5 of the protection decision workflow presented in Section 5.1. While the Protection Enumerator is able to build a set of DPIs by analyzing each asset in isolation, finding the PIs that are suitable to defer the attacks found by the Risk Assessment Engine against the asset security requirement, the solution walker considers the application as a whole, taking into account the complete code structure and the relationships among the various code regions constituting the application: the goal of this algorithm is to produce a comprehensive protection solution, combining a subset of the DPIs inferred by the Protection Enumerator, able to protect the application from all the possible attacks against all the assets comprised in the target program. In doing so, the solution walker considers the interactions among the protections included in the solution, in order to avoid the introduction of inconsistencies that may alter the business logic of the application or even lead to a non-working protected binary. Also, the overhead introduced by the protections comprised in the built solution is taken into account: the ESP user can set limits to the global overheads (detailed in Section 5.4.3) introduced by the application, and the solution walker will output only solutions that stay below this user-defined upper bounds, thus producing a combination of protections that preserve the application user experience (clearly, if global overhead limits set by the user are correct). So, since the ordering among protections is important, the problem the solution walker aims to solve is finding a protection solution, i.e., an ordered set of DPIs, with at least one DPI safeguarding a PO, to protect the whole application, and with an order among DPIs that does not introduce any inconsistencies in the resulting protected binary.

First of all, given the application code structure, and the available DPIs, the algorithm computes the *code correlation sets*: two code regions are if they share at least one line of code. This is necessary, since, as stated in the introduction of this chapter, the relationships among assets must be taken into account, to avoid possible inconsistencies in the resulting solution that may lead to an application not behaving correctly, like in the case of the containment relationship. Another relationship that must be considered is the use of a variable or constant by a code region: for example, if a code region is protected with anti-debugging (see Section 1.1.4), and its execution depends on the value of a string literal that has been encoded with Tigress literals obfuscation (Section 1.1.2), every time that the code moved into the debugger has to access the constant value, a call to the encoder function outside the debugger will be needed to obtain the literal; this will lead to a context switch for each call, which, while not posing a threat for the correctness of application execution, can lead to an unbearable overhead for the application, if the number

of calls is conspicuous, and it is therefore advisable to protect also the string encoder with anti-debugging. Also, using code correlation sets can speed up the solution walker algorithm, since it reduces the number of crosschecks among DPIs in the same solution needed to avoid possible incompatibilities among them. Code correlation sets are built by the solution walker recursively following such relationships among code regions and data: these are found automatically in the first step of the ESP workflow, i.e., with the source code analysis executed by the Source Code Analyzer via CDT (Appendix A). Thus, the problem of finding a suitable ordered combination of protections for all the application assets is divided in finding one ordered combination for each code correlation set: the solution can be therefore redefined as a partially ordered set of DPIs, since only the ordering of DPIs applied to assets in the same code correlation set is important.

In the second step of the workflow, the solution walker chooses, for each PO, the number of DPIs tasked to protect it: a different tuple of integers is generated every time this step is executed, until all the possible tuples are considered. At least one DPI for each PO must be included in the solution, in order to provide at least a minimum protection for each asset in the program. To speed up the computation, the solution space can be reduced by setting a maximum number of DPIs for each PO.

Then, for each tuple of integers, the solution walker executes the third step in its workflow, first building for each PO all the possible sets of DPIs with the length specified for the related PO in the tuple, and then building all the possible combinations of these sets of DPIs, with one set for each PO. Every time this workflow step is executed, one of these combinations of DPI sets is built, and the next step in the workflow is called, until all the possible combinations have been considered. It should be noted that certain protections are *singletons*, i.e., can be applied only once to an asset: an example of singleton protection is code mobility (Section 1.1.3), which protects a code by moving it to a remote server, an operation that obviously can be done only once. Thus, if a DPI employs a singleton protection, the solution walker will apply it at most one time on each asset: this means checking that this DPI is included at most one time among all the sets targeting the POs related to the same asset. To do so, also the containment relationships must be considered, since if for example anti-debugging is used to protect a whole function, also all the code regions contained in it will be moved in the self-debugger: thus, is not possible to apply the same technique on contained code regions, if the container function has been already protected with the same technique.

Then, the solution walker executes the last step in its workflow: given a combination of DPIs protecting all the POs of the application, the solution walker generates all the admissible orderings of for the DPIs, excluding the ones containing *forbidden precedences*, which is cases when applying one protection technique on an application part already protected with another one will lead to a non-working protected binary, such as in the example presented before of the obfuscation of a code already monitored by remote attestation. To reduce the number of different orders that must be considered by the walker, the latter can be instructed to avoid also *discouraged precedences*, i.e., cases when applying one protection to a code or datum already protected by another technique, while not leading to an inconsistency in the resulting binary, will likely have some unwanted effect: for example, obfuscating a code that will be then protected with code mobility is not only completely useless from a security point of view, since the code will be moved to a trusted remote server, but it is also detrimental for the server performances, since the obfuscation will slow down the execution of the moved code. As already explained before, to check that the ordering does not contain any forbidden precedences (and discouraged precedences, if the walker is instructed by the user to do so), the walker needs to verify all the precedences among all the DPIs protecting assets in the same code correlation set. The walker will produce all the possible orderings for the given combination of DPIs; then, the walker will return to the previous step, generating another combination of DPIs to be ordered.

Summarizing, the final result of the walker is a set of valid solutions, able to offer at least

some degree of protection to each PO in the application, and that, if deployed, produce a working protected binary. However, this is not sufficient to deem a solution as effectively being usable to protect the application, since the solution must not penalize excessively the protected application performances. However, the performance overhead introduced deploying a protection on an application asset do not depend solely on the technique used and, but also on the specific characteristic of the protected asset. For example, if the integrity of a code area is safeguarded using remote attestation, the effort needed to compute its hashes, i.e., the additional instructions that must be executed by the CPU of the device running the application due to the deployment of this technique, will be directly dependent on the size of the protected code.

So, to solve both problems, a quantitative measure of the characteristics of the assets is needed, in order to both assess the solution effectiveness in complicating the execution of attacks, and the overhead introduced in the application. ESP solves this problem by using a set of software metrics, presented in Section 5.4, evaluated by the Metrics Framework using DIABLO. The solution walker, after building a solution, calls the Metrics Framework to obtain the global overhead introduced in the application; these will be evaluated by the Metrics Framework using a set of PI-specific formulas (Section 5.4.3) based on the values of the aforementioned metrics.

5.4 Software metrics

This section provides insights on the evaluation of software complexity metrics done by the Metrics Framework, and their subsequent use by the Risk Mitigation Engine. Software metrics can be defined as an "attempt to quantify aspects of a software system" [22]: in particular, complexity metrics try to grasp the difficulty of carrying out tasks on software, for example maintenance, debugging, testing or applying modifications [76]. Indeed, one of those tasks can be attacking a software: therefore, increasing the complexity of a protected program is a desirable result. There are various attempts in literature to leverage complexity measures in order to assess a protection efficacy in defying attacks on a given function or code snippet: a notable example is the *potency*, defined by Collberg in [37], which is a measure of effectiveness for code obfuscation techniques (see Section 1.1.2). Given an obfuscation technique τ that, when applied to a specific program P, produces an obfuscated version P' of the latter, the potency $\tau_{pot}(P)$ of τ in protecting P against reverse engineering can be evaluated with the following formula:

$$\tau_{pot}(P) = \frac{E(P')}{E(P)} - 1 \tag{5.1}$$

Where $E(P)$ is a complexity measure evaluated on the original program P, and $E(P')$ is the same measure computed on the obfuscated program P'. Indeed, an obfuscation technique is effective when $E(P') > E(P)$, since it produces a more complex program that is more difficult to reverse engineer by an attacker w.r.t. the original application. In the aforementioned paper, Collberg proposes seven complexity measures (e.g., size of the program, nesting level of conditional and looping constructs) to evaluate the potency of a transformation; other appropriate measures have been proposed by Tonella *et al.* in [118] (e.g., call graph and data dependency graph size).

As elaborated in Section 5.5.1, the Risk Mitigation Engine leverages the concept of potency to evaluate the effectiveness of a solution in countering the attacks inferred by the Risk Assessment Engine. To evaluate metrics on the unprotected binary involved in the solution assessment, as already reported in Section A.1, the Metrics Framework relies on DIABLO: such metrics on the unprotected program are obtained by calling DIABLO via the ACTC, without applying any protection.

The remainder of this section is organized as follows:

- Section 5.4.1 lists the metrics evaluated by the Metrics Framework on the target application and on its functions;

- Section 5.4.2, mainly derived from the publication "Estimating Software Obfuscation Potency with Artificial Neural Networks" [29], explains how the Metrics Framework predicts values assumed by a function or snippet metrics after the deployment of a specific sequence of protections, without actually applying such transformations to the code;
- Section 5.4.3 presents how the Risk Mitigation Engine, when generating a solution, estimates the impact of chosen protections on the target application performances, should the assessed solution be deployed on the application;

5.4.1 Software protection complexity metrics

ESP supports 21 complexity metrics, as reported in Table 5.1. DIABLO can analyze a binary to calculate metrics on the whole application and for each code region: for ESP, especially the latter kind of metrics are interesting, e.g., to evaluate the effectiveness of a protection applied to a specific asset, taking into account how the asset metrics are changed by the assessed protection. Among those metrics, 18 express generic information about code complexity, while the other five metrics are devised to express the result of specific protections after their deployment to the evaluated code. Also, depending on whether the evaluation of the metrics involves running the program, the following three types of metrics can be identified:

- static metrics: computed without executing the binary;
- dynamic coverage metrics: computed executing the binary, identifying the basic blocks executed, and considering only them in the evaluation of the metrics;
- dynamic size metrics: computed as the coverage metrics, but taking also into account the number of time each basic block is executed.

Metric	Protection	Static	Dyn. Size	Dyn. Coverage
No. of assembler instructions	All	Yes	Yes	Yes
No. of source operands	All	Yes	Yes	Yes
No. of destination operands	All	Yes	Yes	Yes
Halstead's Length (HL)	All	Yes	Yes	Yes
No. of edges in the CFG	All	Yes	Yes	Yes
Cyclomatic Complexity (CC)	All	Yes	Yes	Yes
Control flow indirection metric	All	Yes	Yes	Yes
No. of mobile blocks	Code mobility	Yes	No	No
Total size of mobile basic blocks	Code mobility	Yes	No	No
Size of attestation data	Remote attestation	Yes	No	No
No. of attested basic blocks	Remote attestation	Yes	No	No
Size of attested basic blocks	Remote attestation	Yes	No	No

Table 5.1: Software complexity metrics supported by ESP.

Unfortunately, at the time of writing, ESP supports only static metrics to evaluate the overhead and the protection level introduced by the deployment of a PI on a specific code region or datum in the target application. A first problem for the use of dynamic metrics is that, for DIABLO being able to evaluate dynamic metrics, an example use case of the application is needed from the user: it should resemble as much as possible the expected usage of the application by its users, and should be specified in the ACTC configuration file. ESP is currently not able to automatically

infer the inputs needed for executing the target application: theoretically, it would be possible to integrate in ESP a fuzzer to execute the application various time, and subsequently compute an average of the obtained dynamic metrics; however, there is no guarantee that this random input execution would summarize the typical application usage. This first problem could be nonetheless partially solved by tasking the ESP user to manually specify a use case in the ACTC configuration file. A second problem and still not completely solved, is that, to obtain the metrics of a protected code, the solution must be actually deployed, obtaining the protected binary, so that DIABLO can be launched on it to compute the metrics: this involves applying the protection and compiling the code every time the overhead and the score of a solution must be evaluated: this, especially for large application, cannot be done in a reasonable time, since the number of solutions can be high and the time needed to compile the application is not negligible. Furthermore, the evaluation of dynamic metrics involves the execution of the protected application: again, is not feasible to execute the application every time a solution must be assessed. A partial solution to this problem, presented in Section 5.4.2, is the use of a machine learning approach to estimate the metrics of the protected application, instead of evaluating them via an actual deployment of the protections. However, this approach has been proved effective only to predict static metrics, thus ESP, at least for now, bases its reasoning processes only on static metrics; however, as detailed in Section 2.4, this limitation has not impeded ESP to perform well in protecting real applications.

In the following, details about the specific metrics used are given, starting from protection-independent ones. The *number of assembler instruction* indicates the actual size in the binary corresponding to a code region: for example, this information is useful to indicate the amount of code that an attacker should analyze to understand an asset that should remain unintelligible. The total *number of source and destination operands* represent the amount of different register, memory locations and constants that are respectively read or written by the code: these metrics give another measure of the code complexity, since, to comprehend a code, an attacker should understand the purpose of all the operands involved in the code execution. Halstead's Length (HL) [62] is the arithmetic sum of the three preceding metrics.

Other interesting complexity metrics can be evaluated taking into account the analyzed code region CFG. A first measure is the *number of edges* comprised in the aforementioned graph, since it corresponds to the number of jumps comprised in the examined code, which should be taken into account when trying to reconstruct the code flow to comprehend its purpose. Similarly, McCabe's Cyclomatic Complexity (CC) [88] measures the number of linearly independent paths, from each entry point to each exit point, in the analyzed code region: each path represent a possible different execution of the code region, which an attacker must analyze to ultimately understand the code. Another metric computed on the CFG is the *control flow indirection metric*: it measures the number of indirect edges in the CFG, i.e., the number of jumps whose target is evaluated at run-time as a result of previous calculation in the code (e.g., a jump with the target address in memory written in a CPU register), thus being difficult to be evaluated by an attacker without executing the program.

Regarding protection-dependent techniques, ESP supports five of them. Two are related to the code mobility technique (see Section 1.1.3), and measure the amount of code that is moved to the server, measured either in *number of basic blocks* or in Bytes (*Total size of mobile basic blocks*): since this code is not available in the binary, being download with a per-needed basis at run-time, these metrics indicate the amount of code the attacker will be able to analyze only during execution. The last three metrics are related to remote attestation, an on-line anti-tampering technique described in Section 1.1.3. The *size of attestation data* indicates the size of the attestation measurements evaluated by the technique on the protected code: it has little use for security purposes, but is indeed useful to evaluate the increase in application memory usage caused by the deployment of this technique. The remaining two metrics measure the amount of code monitored by the technique, indicating the amount of code, expressed either as a *number of attested basic*

block or with the total number of Bytes monitored (*size of attested basic blocks*): this measures are interesting from a security point of view, since they express the amount of code that an attacker cannot tamper with, without the technique identifying such changes and executing the proper reaction to modification.

5.4.2 Complexity metrics prediction

When evaluating the effect of a protection solution, both Risk Mitigation Engine and Asset Hiding Engine need to make calculations based on metrics of the code regions protected by the solution: it is needed in particular to evaluate the solution score (see Section 5.5.1 for the score calculated by the Risk Mitigation Engine) and the overhead introduced by the solution (see Section 5.4).

Obviously, the Metrics Framework can evaluate such metrics by actually applying the solution to the code via the Solution Deployer, and subsequently obtaining the metrics on the protected code by calling DIABLO via the ACTC, as it does for the unprotected code metrics: this means compiling the program to obtain the binary on which the static metrics must be evaluated. The Metrics Framework is tasked to do so every time a solution is evaluated; however, all the metrics on all the code regions could be obtained with the above method, testing each possible combination (setting a maximum length) of protections by applying it an all the application code region. Given a program P, and a number n_{pi} of PIs (see Section 3.3) supported by ESP, the time $t_{test}(n_{pi}, L)$ that, using this approach, would be needed to obtain metrics values of all the code regions (see Section 3.2) in each of the possible programs, resulting from the deployment of all the possible combinations of protections $comb(n_{pi}, L)$ comprising at most L protections.

$$t_{test}(n_{pi}, L) = (t_{comp}(P) + t_{metr}(P)) \cdot comb(n_{pi}, L) \tag{5.2}$$

Where $t_{comp}(P)$ and $t_{metr}(P)$ are the times needed to respectively compile and evaluate all the metrics on the given program P. The following rough estimation of the number of combinations is evaluated assuming that all the PIs taken into account can be present only once in the combination, and that all orders of protections in the combination are acceptable[4]:

$$comb(n_{pi}, L) \approx \sum_{l=1}^{L} \left(\binom{n_{pi}}{l} \cdot l! \right) = \sum_{l=1}^{L} \left(\frac{n_{pi}!}{n_{pi}! - l!} \cdot l! \right) \tag{5.3}$$

Considering only the code protections developed in ASPIRE, ESP supports 13 different PIs, which is the value assumed by n_{pi} in the above equation; if the maximum length of combination L is set to three, the number of possible combination of PIs $comb(n_{pi}, L)$ can be estimated as 455; if the time to compile the program and calculate the metrics is estimated in one minute (which would make P a rather simple program), it would take approximately more than seven hours to evaluate all the metrics for all the possible combinations; the needed time increases to almost five days with L set to four.

Therefore, evaluating all the possible metrics with the aforementioned approach is not feasible. To overcome this problem, the Metrics Framework implements an approach based on Machine Learning (ML), a branch of Artificial Intelligence (AI) that encompasses techniques to build applications that can be trained to perform specific tasks, by feeding them with examples of execution of such tasks. More formally, "a computer program is said to learn from experience

[4]Neither of the assumptions hold in reality: some protections, e.g., binary code obfuscations, can be applied more than once to the same code, and there are order of applications of protections that lead to incorrect results (e.g., applying remote attestation and then a protection that changes the attested code would lead the appraiser to believe that the code has been tampered with).

E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [91]. The Metrics Framework, in order to predict the metrics, uses a Machine Learning (ML) technique known as Artificial Neural Network (ANN). Essentially, an Artificial Neural Network (ANN) is a directed and weighted graph, with its nodes (called *artificial neurons*) organized in layers of nodes. In general, each neuron receives a set of inputs, multiplies each of them with a set of neuron-specific weight, giving the result to a neuron-specific *activation function* that produces the neuron output. The ANN inputs are feed to neurons in the first layer, which will produce their output with the activation function; such outputs will be given to as input to neurons in the next layer, with the computation continuing until the final layer is reached, which typically contains only one neuron whose output will be the ANN one. An ANN can be trained to perform a task with a training algorithm (e.g., the backpropagation algorithm [128]), which, given a set of solved instances of the task, i.e., a set of inputs and expected outputs typically called *training set*, is able to adjust the input weights of the ANN neurons and their activation functions. The performances of the trained ANN can be assessed using another set of solved instances, called *test set*, comparing, for each instance of the task, the output produced by the ANN with the expected one.

To solve the metrics prediction problem, the Metrics Framework uses a set of ANNs; each of them is able to predict a specific metric, given all the metrics on a code region, after the application of a specific PI on the given code region. Formally, the prediction of one of these ANNs can be expressed with the following formula, where c^0 is the unprotected code region, c^p is the same code region protected with the PI p, and $m_i(c)$ is i-th metric in the set of n available metrics M_c evaluated on the code region c:

$$f_{i,p}(M_{c^0}) \approx m_i(c^p), M_{c^0} = m_0(c^0), m_1(c^0), \dots m_n(c^0) \tag{5.4}$$

In this way, the effect of a single DPI on the metrics of code region can be assessed. However, a solution may contain more than one DPI applied to the same code region: in this case, the global effect of the solution on the metrics of a code region can be assessed by serially connecting the appropriate ANNs. For example, if a code region c is protected with a PI p_1, and then with a PI p_2, a metric $m_i(c^{p_1,p_2})$ can be predicted as follows:

$$f_{i,p}(M^*_{c^{p_1}}) \approx m_i(c^{p_1,p_2}), M^*_{c^{p_1}} \approx m_0(c^{p_1}), m_1(c^{p_1}), \dots m_n(c^{p_1}) \tag{5.5}$$

With the metrics in the set $M^*_{c^{p_1}}$, given in input to the ANN, have been predicted in turn with the appropriate ANNs with the unprotected c metrics given in input:

$$M^*_{c^{p_1}} = f_{0,p}(M_{c^0}), f_{1,p}(M_{c^0}), \dots f_{n,p}(M_{c^0}) \tag{5.6}$$

Each ANN has been trained with metrics evaluated by DIABLO on all code regions in the source code of 21 FOSS packages in the Linux Debian[5] distribution repository, listed in Table 5.2, obtaining all the metric before and after the application of each PI on all the available code regions. DIABLO has been called on the object files of such program, previously compiled with GCC configured with three different optimization levels (`-Os`, `-O2`, `-O0`). In particular, the obtained code regions where 35510, of which 90% has been used for training the ANN, and the remaining 10% to test them.

The obtained ANNs perform well, with a coefficient of determination[6] R^2 over 90% for every protection technique. For example, Figure 5.2 shows the error plot for the predictions, on all

[5] https://www.debian.org/

[6] https://it.mathworks.com/help/stats/coefficient-of-determination-r-squared.html

PACKET	VERSION	CODE REGIONS			
		-Os	-O2	-O0	TOTAL
bc	1.06.95	495	462	520	1477
bzip2	1.0.6	186	165	225	576
ccrypt	1.10	158	140	156	454
dash	0.5.8	1171	1079	1229	3479
ed	1.10	226	205	245	676
findutils	4.7.0	1165	1178	1393	3736
flex	2.6.1	965	859	1041	2865
libdvdread	5.0.3	524	481	553	1558
libjpeg	8d	1041	851	1571	3463
libnet	1.1.6	576	586	563	1725
libstarlink-pal	0.5.0	1137	1144	1265	3546
netcat	1.10	71	61	60	192
qmail	1.06	149	131	122	402
time	1.7	38	31	41	110
udo	6.4.1	2243	1998	2273	6514
unrtf	0.21.9	455	437	459	1351
vnstat	1.15	420	348	417	1185
watchdog	5.15	282	244	293	819
wbox	5	81	86	95	262
whitedb	0.7.3	154	123	155	432
zenlisp	2013.11.22	247	191	250	688
TOTAL		11784	10800	12926	35510

Table 5.2: Code regions in the metrics prediction data set.

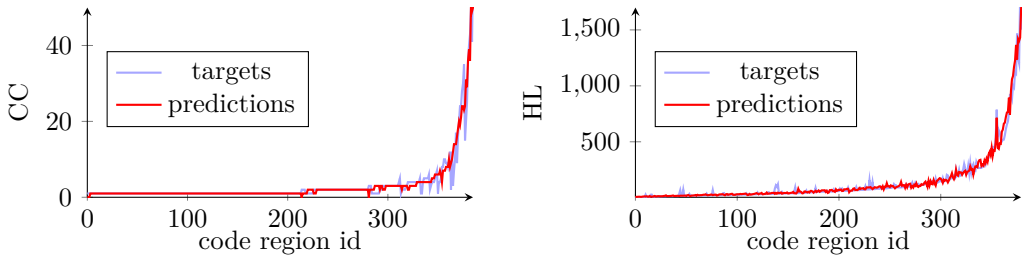


Figure 5.2: CC and HL predictions for "branch functions, high overhead" PI.

available code region in the test set (ordered by metric unprotected value), of the CC and HL metrics after the application of branch functions with DIABLO (see Section 1.1.2). It is interesting noticing that the line related to the real HL metrics values line show a numerous number of spikes, which are well approximated by the predicted values line.

5.4.3 Protection overhead estimation

When protecting a software, the detrimental effect of the applied techniques on its performances must be taken into account, in order to obtain a software that, while exposing a good resistance against possible threats to its asset, has still an acceptable user experience. Indeed, since in general protection techniques add instructions, increase the complexity of a program CFG with convoluted code constructs (e.g., the large number of jumps needed to flatten a code), or exchange messages over the Internet (e.g., remote attestation requests and responses), a certain performance decrease is an inevitable effect of software protection.

Measuring the *overhead* of a solution is therefore essential for the Risk Mitigation Engine: while it may be great in protecting the application assets against the attacks inferred by the Risk

Assessment Engine, its deployment by the Solution Deployer could result in a protected binary whose execution is practically unfeasible, being too slow, or occupying too much memory on the device used do run the program. The same problem is faced by the Asset Hiding Engine, which deploys protection on non-asset code regions: it could theoretically protect the whole program, but is constrained in the number of applicable protection by the overhead introduced by the latter.

To evaluate the overhead caused by the deployment of a protection on a specific code region, the Metrics Framework uses a set of protection-specific formulas, obtained during the ASPIRE project by the protection techniques developers. The formulas are essentially linear combinations of complexity metrics evaluated after the deployment of the specific protection. When assessing the overhead introduced by a solution, ESP evaluates it for each code region protected by the solution. In particular, ESP takes into account five types of overhead:

- client CPU overhead: the percentage of additional instructions executed by the client device CPU when executing the protected code region, compared with the instructions executed by the same device CPU when running the original code;
- server CPU overhead: the percentage of additional instructions, constituting the routines of the assessed protection in the ASCL (see Section 1.1.3), executed by the server CPU in order to service the requests made by the client due to the deployment of an on-line protection, w.r.t. the number of ASCL instruction executed without servicing any client protection requests (i.e., with no on-line protections deployed to the server);
- client memory overhead: the amount of the client device memory needed to store the data needed by the protection, measured in Bytes;
- service memory overhead: the amount of the server memory needed to store the data needed by the protection service routines in the ASCL, measured in Bytes;
- network overhead: the average amount of bandwidth occupied on the client device Internet connection, during the application execution, due to messages exchanged by the on-line protection technique with the ASCL.

In the protection decision workflow (Section 5.1), the Metrics Framework is tasked to assess the global overheads introduced into the target application by a solution, after its deployment on the target: this is needed to ensure that such overheads are limited[7], so that the produced binary does not present a performance degradation so conspicuous that the application becomes unusable. The Metrics Framework can predict the metrics of the assets, which are modified by the deployment of the solution, using the neural networks described in Section 5.4.2, and use such predictions to estimate the values of the overheads, by feeding the predicted metrics to the overhead formulas presented above in this section. This leads to the estimation of overheads for each asset: however, a global overhead value is needed, summarizing the medium slow-down introduced by the protections, in order to assess the feasibility of the solution. The Metrics Framework obtains a global value for each kind of overhead, combining the relative overhead values computed on each function comprised in the application source, using the following weighted sum where each overhead is first multiplied to a weight, indicating the percentage of instructions of the related function w.r.t. the total number of instructions constituting the application code. It should be noted that also the overheads introduced by protections on variables are considered: since these protections introduce more code in the function containing the protected variable, such as for example the additional instructions needed to decode variables obfuscated with the technique

[7]The overhead global limits for the protected target application must be set by the ESP user.

described in Section 1.1.2, the metrics of the function will be impacted by such protections, and can be used to assess the overhead introduced by such techniques as it is done for their code counterparts.

5.5 Asset protection solution inference

This section presents the solution solver algorithm, which is tasked to decide, among the asset protection solutions inferred by the solution walker algorithm (see Section 5.3), the ones that are best suitable to deferring the possible APs that can endanger the POs, i.e., the user-defined security requirements of the application assets. In doing so, the solver assigns to each evaluated solution a score, called protection index, whose computation is described in Section 5.5.1.

As already described in the introduction of this chapter, this problem is not trivial: the relationships between the protected assets code structure and security requirements, the possible attacks against them, the deployed protections with the interactions among them must all be modelled in order to find a quantitative way to score the solutions, so that they can be compared and the best solutions, i.e., the ones with most chances to effectively defer the attacker, can be presented to the user for subsequent deployment on the target application. The basic idea behind the solution to this problem is modelling the protection decision process in a MATE scenario as a game, with characteristics similar to chess. The checkerboard is the application code, the white player (who moves first) is the defender tasked to produce a protected version of the application that must be publicly released afterwards, and the black player is the attacker that must breach at least an asset security requirement. The defender moves its pieces, i.e., the protections, by deploying them on the application; the attacker moves are instead executions of attack paths against the protected application. It should be noted that, due to the characteristics of the MATE scenario, there is one fundamental difference between the chess and the protection decision game: in chess, player moves are alternated, while in a MATE scenario, the defender has the first move, protecting the application before its release, but afterwards the attacker has theoretically infinite moves. Thus, at the protection decision game the defender wins if it defers as much as possible the breaching of assets security requirements, choosing the combination of protections most effective in increasing the effort needed to carry out successful attacks: since perfect protections do not exist (this has been proven formally for obfuscation techniques by Barak *et al.* [11]), an attacker with infinite time (and subsequently moves) will, in the end, be able to break all the protections chosen by the defender. However, a well-protected application may cause the attacker to throw in the towel after some failed attack attempts. Also, the economic model behind the attacker willingness to breach the application may, in fact, limit the time: for example, an attacker may want to break a licensing scheme of a commercial application, in order to build a crack to use it without a proper license, and thus sell the crack on the black market; however, if a new version of the application is released before the attacker can succeed in building the crack, even if it in the ends succeeds in creating the crack, it would be incompatible with new application version, and thus would not have any value on the black market (or at least a greatly reduced one).

Using this parallel with chess, all the abundant research on the automation of this game can be adapted for protection decision purposes. This research has its roots in the broader field of game theory, which, as defined by Myerson in [93], is *"the study of mathematical models of strategic interaction between rational decision-makers"*, and has been applied to solve problems in many fields, for example in economy, e.g., to model strategies of companies [110], or for military purposes, e.g., to analyze possible choices for nuclear deterrence during the cold war [25]. Indeed, research has been also on possible applications of game theory for cybersecurity, as presented in a recent survey [43] by Cuong *et al.*: reviewed works include uses of game theory to model attack and defence of smart grid systems, computer networks and cryptographic schemes.

Building computer programs able to play chess, a problem known as *chess programming*, has

been first introduced by Shannon in 1949 [34]. In this seminal paper, two of the core concepts of chess programming are applied to this problem for the first time: the mini-max algorithm (presented in 1928 by Borel in [24]), used to model the evolution of a game in various states (e.g., different disposition of pieces on the checkerboard) through moves of the players, and the use of an evaluation function, i.e., an heuristic to estimate the probability of a player winning a game in a specified state. These concepts are still at the base of modern chess engines, such as IBM Deep Blue [28], a special-purpose computer that in 1996 was the first one able to win a chess game against a human chess world champion, Garry Kasparov, and Stockfish[8], an open-source engine which at the time of writing is at the top of the *CCRL 40/4* chess engine ratings[9].

For zero-sum games like chess, where the gain resulting from one player's move is exactly the same as the damage for the other player (so that their sum is zero), a single score for a position can be used, ranging between -1 (black player's checkmate) and $+1$ (white player's checkmate). Thus, given a game state, if it is the white player turn to move, it will try to find a move that leads to another game state with a score higher than the precedent one; the opposite applies if it is the black player's turn. The same holds for the protection decision game: the defender's protections will increase the score, while the attacker will execute APs to decrease it. For chess, the score of a state is linked to various factors, for example the number of pieces still in game for each player, their location on the chessboard, and if the players have performed castling. In Shannon's paper, the following example evaluation function is provided, in order to evaluate the likelihood of a white player winning a chess game in a state P:

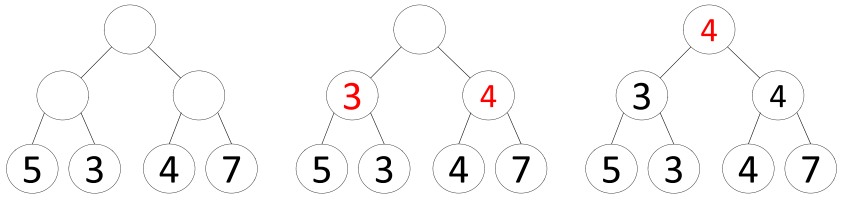
$$\begin{aligned} f(P) =& 200(K-K') + 9(Q-Q') + 5(R-R') + 3(B-B'+N-N') \\ &+ (P-P') - 0.5(D-D'+S-S'+I-I') + 0.1(M-M') + ... \end{aligned} \tag{5.7}$$

in which K,Q,R,B,B,P indicate respectively the number of white kings, queens, rooks, bishops, knights and pawns still in play; D,S,I are penalties for doubled, backward and isolated white pawns; M is a bonus for the mobility of white pieces, i.e., the number of legal moves available to the white player; the primed versions of this variables indicate their relative black pieces counterparts.

The mini-max algorithm implements a worst-case analysis: when the opponent has to move, it will always choose the best move, i.e., the one that, given the current state, will lead to a new state that is the most favorable for him or her; translated to chess, if the opponent is the black player, he will choose every time the move that leads to the state with the lowest score possible. Thus, since the opponent will choose, among the moves available, the ones that maximize the loss for the player, the mini-max algorithm chooses the move that leads, after the opponent move, to the minimum of the losses maximized by the opponent. Figure 5.3 sketches the reasoning process of a mini-max algorithm, trying to find the best move for a player that must maximize the score; due to computational limitations, the algorithm is able to foresee only the possible outcomes after two moves, the one made by the algorithm and the next move made by the opponent. The figure renders the mini-max reasoning process using a tree: each node is a game state, with the root the initial one, while the edges are the possible moves, each one leading from one state to another. The algorithm starts to compute the scores associated with the possible outcomes. Then, it assigns a score to each node in the layer immediately above the leaves (Figure 5.3a): having the nodes only three layers, these are the possible states in which the opponent must move, immediately after the player move that is subject to evaluation by the mini-max algorithm. Using a worst-case analysis, the algorithm infers that the opponent will choose the move that minimizes at most the score: so, each node in the opponent layer will be associated with the lowest score among

[8] https://stockfishchess.org/

[9] http://ccrl.chessdom.com/ccrl/404/



(a) Step 1. (b) Step 2. (c) Step 3.

Figure 5.3: Example execution of a mini-max algorithm.

its immediate successors (Figure 5.3b), which means assigning 3 to the left node in the second layer, and 4 to the right node. Then, the process is repeated, but this time the player must move, so each node (in the figure only the root) will be assigned with the maximum of the scores of its children (Figure 5.3c). In this way, the move is made taking into account also the reaction of the opponent; clearly, the number of forecast moves can be higher, with various alternating minimizing and maximizing phases, but the principle behind the algorithm holds.

The mini-max algorithm and the evaluation functions are the foundations on which the solution solver algorithm is built.

5.5.1 Protection index

To build the solution solver mini-max algorithm, first a score for its states must be defined. As said before, the protection decision process is modelled as a zero-sum game: there is a unique score for the states in the mini-max tree, representing the global level of security of the application, i.e., the effort needed by an attacker to breach the user-defined security requirement of the assets in the application. Indeed, modelling this effort is not trivial, since it depends on a series of factors. First, the profile of the attacker must be taken into account: for example, a seasoned attacker, proficient in the assembly language, will need considerably less effort to understand the purpose of an algorithm in the binary, in respect with a newbie that has never seen assembly before. However, if an attack simply involves using an automated attack tool, the effect of the attacker profile on the difficulty of carrying out the attack may be less broad. Also, the inherent characteristics of the assets must be taken into account, but also the relationships among them: intuitively, comprehending a code with a complex CFG is harder than understanding a simple code with few basic blocks; however, this holds also when an attacker wants to understand the purpose of a local variable in a function, a task that will be considerably harder if the variable is accessed in various points of a complex function comprising many instructions in a convoluted CFG.

Thus, a set of measures to model the security level of the application are needed, in order to grasp all these nuances. Protections deployed on the application must increase at least one of the measures, while successful attacks will decrease one or more measures. The solution solver, to evaluate the *protection index*, i.e., the score of mini-max tree leaves, uses the following measures, computed for each asset:

- *uncertainty*: the confusion an attacker feels when trying to understand the purpose and logic of an asset, for example an obfuscated one (Section 1.1.2);
- *remote instructions*: the increased difficulty an attacker encounters, when parts of the target application code are not under his control, having been moved to a safe remote location, e.g., using code mobility (Section 1.1.3);

- *alteration detection*: the capability of a protected application to detect when the attacker is attempting to tamper with it, increasing the difficulty of these types of attacks, since the attacker must tamper the protected assets without tripping off any alarm, e.g., avoiding the detection enforced by remote attestation (Section 1.1.3);
- *alteration avoidance*: the ability of a protected application to negate a modification, for example blocking tools used by the attacker to tamper with the application code, as it is done by the anti-debugging technique (Section 1.1.4).

The attacker will try to lower the protection index, by executing attacks that impact one or more of these measures. For example, an attacker that sniffs the traffic of an application protected with code mobility, has a good chance to identify the remote instructions downloaded by the application: thus, if an AP containing a sniffing Attack Step (AS) is played by the attacker, the *remote instructions* metrics is lowered accordingly. Indeed, the profile of the attacker has an impact on the probability of success of this attack: while sniffing the traffic is not complicated (using for example a free tool like Wireshark[10]), understanding that the application is downloading the instructions, and consequently identify them in the traffic captures, is definitely more difficult, and requires knowledge of the protection technique by the attacker. Another example is trying to understand of an obfuscated asset code, thus having an high uncertainty measure: the attacker may resort to static or dynamic analysis tools, in order to obtain more understandable representation of the code (e.g., CFG, execution traces), which will, in turn, decrease the uncertainty of the attacker regarding the logic of the asset code. The attacker expertise can be set by the user via the User Interface. Indeed, knowing the possible type of attacker is fundamental when the protections are decided: if the target program is a mobile app, sold for a low price, protecting it with heavy measures is useless, since the reward of breaking the application will be probably not enough to interest an experienced attacker; the only result would be an unnecessarily slow application, and, if proprietary protections are taken into account, the cost of their licenses could exceed the profits gained with application sales.

Attacks and protections try respectively to breach and defend the assets security requirement: this link is embedded also in the measures: each of them specifies also the risk of the attacker being able to breach a specific security requirement. Uncertainty is strictly related to confidentiality, since for example an attacker that becomes certain of the logic of a code has succeeded in reverse engineering it. Similarly, remote instructions of a code, moved to a trusted server out of the attacker reach, have more possibility to remain confidential. Alteration detection and alteration avoidance are clearly related to integrity and execution correctness[11], since they express the capability of an application to detect or stop attacker attempts to tamper with the protected assets.

Also, as explained before, these measures must be linked to the characteristics of the asset code: therefore, they are linked to the software metrics (Section 5.4) of the asset. In particular, the initial values of these measures, after the single defender move (e.g., the deployment of one of the solutions found by the walker) are evaluated as weighted sums of the software metrics. Thus, the protections increase the protection index because they increase the metrics: this behaviour takes its roots from the concept of transformation potency proposed by Collberg, as explained in Section 5.4. For example, the uncertainty measure of an asset is based on the HL and CC of its code, two measures that are increased by code obfuscation techniques (e.g., the CFF technique

[10] https://www.wireshark.org/

[11] The execution correctness is a stronger form of integrity for code, leading to complex attack paths that require more effort for the attacker, as explained in Chapter 4.

presented in Section 1.1.2 increases the CC, since the resulting CFG contains a high number of parallel paths).

Given a target application A, its global protection index $P(A)$ in a state of the mini-max algorithm is evaluated by the solution solver using the following formula:

$$P(A) = \sum_{i}^{\#assets} w_i \cdot \left(\sum_{j}^{\#measures} \beta_j \cdot s_{ij} - L \cdot l_{ij} \right) \tag{5.8}$$

where w_i is a user-defined weight representing the importance of the i-th asset, s_ij is the value of the j-th security measure evaluated on the i-th asset, and β_j is a user-defined coefficient for the j-th security measure. The attacker succeeds in violating the i-th asset if at least one measure, after the successful execution of one or more APs against the asset, goes below a user-defined constant threshold T; this is expressed in the formula with a user-defined constant L and the l_{ij} boolean variable, evaluated as following:

$$l_{ij} = \begin{cases} 1, & \text{if } s_{ij} < T \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

Since the attacker wins if he succeeds in breaching at least one security measure (i.e., one asset security requirement), the L constant should be set to a very big value, so that the mini-max algorithm will avoid selecting moves that lead to such losing states. Also, if the user has not set a security requirement for an asset (e.g., it is not important that a function is understood by an attacker, as long as is not able to modify it), the related measures s_{ij} are excluded from the formula.

Finally, a way to quantify the impact of successful APs on security measures is needed, so that the mini-max algorithm can update the score accordingly. This impact of an AP is assessed taking into account the inherent difficulty of ASs constituting the AP and the probability, given the attacker expertise, of successfully execute such ASs, but also the ability of protections in thwarting specific types of attack tools, and the increased resistance against specific AS due to synergies among protections. All this information, stored in Knowledge Base, derive from experts knowledge gathered during the ASPIRE project. When the attacker plays an AP, each of the constituting AS is executed following the order specified by the AP, updating the measures of the affected asset. In particular, after the execution of the k-th AS in the played AP, targeting the i-th asset, the j-th measure will take the following value:

$$s'_{ij} = \big(1 - (\pi_k \cdot (1 - \omega_k) \cdot \lambda_k)\big) \cdot s_{ij} \tag{5.10}$$

where:

- s_{ij} and s'_{ij} are the impacted measure values, respectively before and after the assessed attack step execution;
- $0 \leq \pi_k \leq 1$ is the attacker probability of successfully execute the k-th AS, depending from the type of AS and the selected attacker expertise;
- $0 \leq \lambda_k \leq 1$ assesses the impact of the AS type against the security measure (e.g., dynamic analysis, carried out for example using a debugger, as an high impact on the confusion measure);
- $0 \leq \omega_k \leq 1$ represent the mitigation effect of protections applied on the i-th asset against the type of AS.

The global mitigation effect ω_k of the DPIs applied on the i-th asset against the k-th AS in the assessed AP, used to evaluate s'_{ij}, can be obtained by combining the mitigation effects of each DPI applied to the i-th asset, as in the following formula:

$$\omega_k = 1 - \prod_{p}^{\#\text{DPIs on i-th asset}} (1 - \omega_{k,p}) \tag{5.11}$$

The protection mitigation effect ω_k must take into account also the ordering of protections in the solution played by the defender, with a bonus for encouraged precedences and a malus for discouraged ones (Section 5.3), and is obtained by combining the mitigation effects of each DPI applied to the i-th asset. Given the p-th DPI applied on an asset endangered by the k-th AS in the AP, the relative $\omega_{k,p}$ mitigation effect is obtained with the following formula:

$$\omega_{k,p} = 1 - (1 - m_{k,p}) \cdot \frac{(1-E)^e}{(1-D)^d} \tag{5.12}$$

where:

- $0 \leq m_{k,p} \leq 1$ is the level of mitigation introduced by the p-th protection against the k-th AS, if p is the only protection applied to the endangered asset;
- $0 \leq E \leq 1$ and $0 \leq D \leq 1$ are respectively the user-defined bonus for encouraged precedences and malus for discouraged ones;
- e and d are respectively the number of encouraged and discouraged precedences, i.e., the number of DPIs, applied on the same asset as the p-th one, preceding the latter in the solution, which are respectively favoured or hindered by the deployment of the p-th DPI.

Default values for the various constants used by the solution solvers have been found empirically, assessing the solutions found by the Risk Mitigation Engine on the ASPIRE use-case applications w.r.t. solutions devised manually on such applications by experts of the software security companies involved in the aforementioned project (Section 2.4).

5.5.2 Solution solver mini-max algorithm

The solution solver uses a mini-max algorithm in order to find the score for each solution built by the walker (Section 5.3), assessed in a worst-case scenario, i.e., when the attacker chooses the best AP to breach at least one security requirement of an asset in the lowest possible time. Figure 5.4 sketches a mini-max tree built by the solution solver: the root represents the best solution score found so far, the first layer contains the solutions built by the walker, and the remaining layers the possible AP mounted by the attacker against the application assets, given the solution chosen by the defender. It should be noted that an AP can be repeated an infinite number of times by an attacker (i.e., an attacker can concentrate on a specific attack for a longer time), thus the tree is theoretically infinite: the mini-max algorithm stops after a user-defined number of layers in the mini-max tree, i.e., number of APs tried by the attacker, such as a chess engine forecasts a limited number of moves; clearly, a deep search will result in more precise scores, but will take a longer time for the solver to produce them. More precisely, the time needed for the execution of the solver grows exponentially with the number of layers in the tree: if the Risk Assessment Engine infers a APs against the application assets, the user sets a number l of mini-max tree layers, and the solution walker builds s different solutions, the mini-max tree will contain $s \cdot a^l$ states, whose score must be evaluated using the formulas described in Section 5.5.1. Thus, times needed to evaluate the solution scores can become unfeasible for bigger applications,

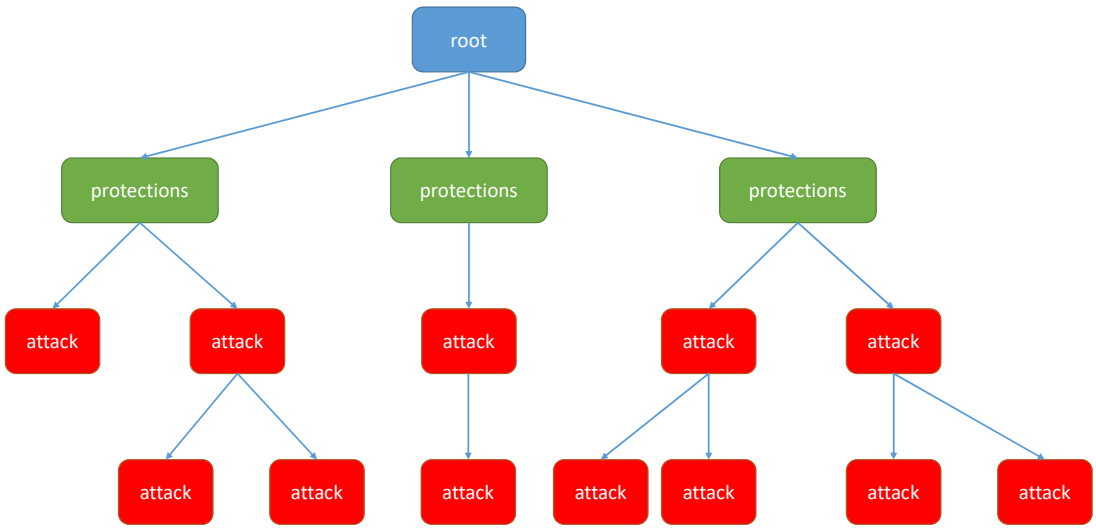


Figure 5.4: Example of a mini-max tree built by the solution solver.

containing many assets: the solution walker mini-max implementation uses a set of strategies to reduce the number of the states that must be evaluated; in fact, the tree in figure is unbalanced, since some of the possible states are not considered due do these strategies. First of all, the code correlation sets, described in Section 5.3, are used to split the trees: since an AP against an asset can impact only the measures of the targeted assets, and of the other ones contained in the same correlation set (e.g., if the attacker debugs a function, it does automatically the same for all the code region contained in the function), a separate smaller tree can be used to model sequences of APs targeting the same code correlation set; due to the exponential nature of the mini-max algorithm, this greatly reduces the computation time.

Also, the following standard enhancement strategies, typical of chess programming (used by all the modern chess engines), are implemented to further speed-up the solution solver:

- *Alpha-Beta pruning*: a branch-and-bound technique, where α and β are respectively a lower and higher bound; if a move leads to a state with a score lower than α or higher than β, it is considered respectively too bad (a better move surely exists) or too good to be true (due to the worst-case scenario considered, the attacker will never choose the moves that lead to this state), thus the mini-max does not consider it anymore, pruning from the tree this state and all its children;

- *Reduction*: a technique that avoids further analyzing a state, if the outcome of the subsequent moves is already clear; for example, if a state contains a security measure that is almost broken, it is clear that the attacker will succeed in breaking with few more APs, thus winning the game, and therefore the solution solver will deem the state as undesirable without further searches (i.e., not considering possible children);

- *Transposition table*: a large hash table, containing information about a state already encountered before, but with another sequence of moves (a *transposition*, in chess jargon).

This techniques need a set of user-defined parameters (e.g., α and β for pruning), whose value can greatly affect the quality of the found solution, but are difficult to find: however, the user can launch the mini-max algorithm with a shallow depth, obtaining a solution in no time, in order to assess a medium value for the solution score, and base the parameters on this medium value. For example, the optimal solution score must be comprised between α and β for the mini-max algorithm being able to find it: so, if a fast execution leads to a best solution with score 50, probably the optimal solution score will be near that value, thus α and β can be set accordingly, e.g., to 40 and 60. Then, the solution solver can be launched with a deep search, which will lead to the optimal solution (or at least to an approximation of it).

5.6 Validation

This section compares the asset protection methodologies described in this chapter with the ESP requirements detailed in Section 2.1.

Regarding the risk framing requirements (see Section 2.1.1), they are completely satisfied by the ESP asset protection phase. First, attacker skills are taken into account in the solution score computed by the solution solver algorithm (see Section 5.5). Also, the protection solutions are presented to the user in a detailed and human-readable format, specifying for each user-defined asset the protection techniques that must be used to safeguard each of its security requirements, the protection tool that must be used to deploy the technique to the asset, along with the configuration parameters needed to drive the tool in this process, and the order of application of each protection to the asset. Finally, the algorithms that implement the asset protection methodologies described in this chapter can be fully customized by the user, with a set of parameters that influence the time needed to execute them, and the accuracy of the produced results. A deep exploration of the solution space may be performed prior to the distribution of the application to the public, thus obtaining a solution near the optimum. Conversely, a fast solution search may be performed each time a patch for the software must be released, when time is an issue, thus refining the existing protection solution with additional protections to defer possible attacks enabled by the introduction or modification of code performed by the patch. Thus, the asset protection phases adapt to the software life-cycle.

Regarding the risk mitigation requirements (see Section 2.1.3), they are satisfied by the methodologies presented in this chapter. The main objective, deferring the possible attacks against the application assets, is completely accomplished, covering all the attack paths inferred during the risk assessment phase of ESP (see Chapter 4). The protection solutions are obtained taking into account the application structure and the complexity metrics of the code comprising the user-defined assets. Furthermore, ESP supports automated protection tools for each of the techniques employed in the asset protection phase, thus the deployment of the obtained protection solutions can be completely automated. Overheads are also taken into account when the solutions are inferred, thus the user experienced is not hampered, with protected binaries obtained after the deployment of such solutions still responsive to the user.

Requirements related to risk monitoring (Section 2.1.3) are respected, since the solution score is evaluated by the solution solver (see Section 5.5) simulating a penetration testing on the application protected with the evaluated solution, with a game-theoretic approach that employs the attack paths inferred in ESP risk assessment phase to evaluate the resistance against threats gained by assets after the deployment of the protection in the evaluated solution on the target application.

In the validation phase of the ASPIRE project, the experts analyzed the protection solutions inferred by ESP in this phase on the project use-case applications. They were pleased by such solutions, deeming their deployment to the applications effective in deferring possible attacks against the comprised assets. Furthermore, they evaluated the solution as free from inconsistencies that

would lead to non-working applications. The overheads introduced were considered acceptable, with a limited impact of the protections employed on user experience.

Chapter 6

Asset hiding

> All warfare is based on deception.
>
> Sun Tzu

This section describes the ESP asset hiding phase. Its main objective is to mitigate a side-effect of software protections, the introduction of *fingerprints*. This term encompasses various characteristics of protected code that an attacker can identify in the binary, thus being able to quickly find the assets in the application. The asset hiding phase refines the protection solution, i.e., the combination of protections, inferred in the ESP risk mitigation phase, deploying additional protections on areas of code not marked as assets by the user, thus aiming to confuse the attacker and leading him or her to focus its attack on these code areas that are not sensitive from a security point of view.

Additional protections are inferred by ESP with the following workflow:

1. *MILP model instantiation*: given an asset protection solution (see Section 5.5) and the target application source code structure (see Section 3.2), ESP infers a Mixed Integer-Linear Programming (MILP) problem tailored for the aforementioned code structure, modelling the attacker confusion due to the additional asset hiding protections;

2. *MILP model solving*: an external MILP solver (at the time of writing, ESP supports IBM ILOG CPlex[1] and lpsolve[2]) is tasked with the solution of the model built in the previous phase of the workflow;

3. *MILP solution translation*: given the model solution obtained in the previous phase, the corresponding additional protection solutions are inferred and used to refine the original asset protection solution.

This chapter is a reworked version of the publication "Towards Optimally Hiding Protected Assets in Software Applications" [105], and is structured in the following sections:

- Section 6.1 presents the concept of protection fingerprints, describing also the characteristics of the fingerprints introduced by the protection techniques supported by ESP;

[1] https://www.ibm.com/analytics/cplex-optimizer

[2] http://lpsolve.sourceforge.net/

- Section 6.2 contains an high-level description of the strategies employed by ESP to decide the additional protections aiming to hide the asset protection fingerprints;
- Section 6.3 details the MILP problem that formalizes the decision process of the additional asset hiding protections;
- Section 6.4 describes how ESP generates a refined protection solution, starting from a MILP problem instance solution;
- Section 6.5 reports on how this additional phase of the ESP workflow addresses the related security requirements detailed in Section 2.1.

6.1 Protection fingerprints

One of the first activities performed by attackers, when they target an application, is identifying the assets in the application binary, in order to focus their attacks on smaller areas of code. This attacker behaviour has been verified empirically, with a study performed on attack reports of professional white-hat hackers [31].

A protection fingerprint is a peculiar characteristic assumed by a code area of the application, after the deployment of one or more protection techniques to it, which can reduce the effort needed by the attacker to locate the protected code area in the binary. Thus, when an attacker targets a protected application, he or she can look for such fingerprints, and can consequently find the assets in the application. In this way, the attacker can focus his or her efforts on the assets code.

Fingerprints can be subdivided among static and dynamic ones. The first kind of fingerprint can be observed by an attacker using static code analysis tools and code representations, e.g., the CFG of the disassembled application. The second kind of fingerprint is instead noticeable by the attacker during the application execution, examining the latter with a debugger or by analyzing execution traces, collected during execution, after the program termination.

A first example of static fingerprint is the one introduced into the CFG of code areas protected with Control Flow Flattening, a code obfuscation technique described in Section 1.1.2. Indeed, the protected CFG assumes a peculiar structure, with a loop with a nested switch statement, with a high number of parallel paths in the CFG. A depiction of a CFG protected with CFF is depicted in Fig. 6.1.

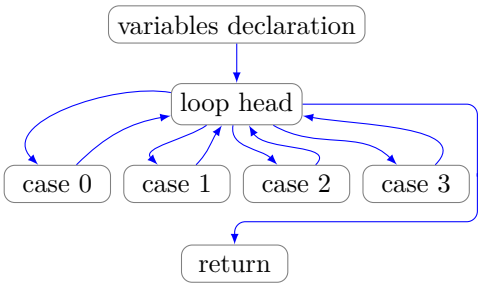


Figure 6.1: Control flow graph before and after the control flow flattening obfuscation.

Another kind of technique that exposes an evident static fingerprint is virtualization obfuscation, described in Sec. 1.1.2. This technique is based on the translation of the code that must be protected into a bytecode, interpreted and executed by a virtual machine specially designed for the protection. Static analysis technique may fail when applied to code protected with such techniques: for example, opcodes of the virtual machine instructions may be invalid for the CPU

technique	replication	enlargement	shadowing
anti-debugging	HIGH	HIGH	–
branch functions	LOW	HIGH	control flow flattening (HIGH), opaque predicates (LOW)
code mobility	HIGH	HIGH	anti-debugging (HIGH), branch functions (LOW), control flow flattening (HIGH), opaque predicates (LOW), virtualization obfuscation (HIGH)
control flow flattening	HIGH	HIGH	opaque predicates (LOW)
opaque predicates	HIGH	HIGH	control flow flattening (HIGH)
virtualization obfuscation	LOW	LOW	anti-debugging (HIGH), branch functions (LOW), control flow flattening (HIGH), opaque predicates (LOW), virtualization obfuscation (HIGH)
static remote attestation	LOW	HIGH	–

Table 6.1: Asset hiding strategies for the protection techniques employed by ESP.

architecture, thus leading a disassembler to produce invalid instructions for the code areas protected by the technique. This failure may arise suspect in the attacker, which may be lead to further investigate such protected areas.

Finally, protections that introduce calls to external libraries can be suspicious for the attacker. For example, on-line protection techniques, such as the one described in 1.1.3, need to either use external network APIs or the native socket system calls of the OS. In both cases, the attacker may recognize such calls: especially for applications that do not need a network connection for their business logic, this may lead the attacker to analyze these protected areas of code. Being observable by analyzing the application binary code, fingerprints of this kind are static.

However, due to the remote connections established at run-time, the on-line protection techniques exhibit also dynamic fingerprints. For example, applications protected with code mobility (see Section 1.1.3), when their execution flow reaches a code area moved to the remote server employed by the protection technique, need to download such code from the aforementioned server: also in this case, the opening of such connections, again especially in applications that would not need a remote connection, if not protected with this technique, can raise suspicion in the attacker, and may lead him or her to analyze the code opening such connection in more detail. Also, the attacker may try to eavesdrop the suspicious traffic, thus recovering the asset code moved to the server, if an encrypted connection is not employed.

Also, code obfuscation techniques introduce dynamic fingerprints on protected code. For example, opaque predicates (see Section 1.1.2) may introduce paths in the CFG that are never traversed by the program flow. An attacker performing a dynamic analysis of the binary, for example with code coverage tools, may identify such paths never taken, and may analyze them, finding the embedded opaque predicates. A methodology for automatically identify and remove opaque predicates in code have been proposed by Dalla Preda *et al.* [45].

6.2 Asset hiding strategies

As reported by software security experts involved in the ASPIRE project, the main solution in the software protection industry to defer the identification of assets by attackers is applying obfuscation to a large part of the program code, protecting also areas of code that do not contain

assets. However, this process is performed manually and can be time-consuming. The protected application must be tested to ensure that the additional obfuscation of code has not introduced an excessive overhead in the application, thus the obfuscation and application testing process can be repeated multiple times, until an optimal trade-off between security and overhead is reached. Thus, automating this process, by designing ad-hoc strategies to hide the location of the assets in the binary from attackers, could be useful for security experts.

This section presents the three *asset hiding strategies* designed for this purpose: fingerprint replication, enlargement and shadowing.

The *fingerprint replication* involves the insertion in the application of additional fingerprints, devised to have a similar structure to the ones introduced in the ESP asset protection phase. In practice, this is obtained by applying the same protections used in the aforementioned phase on code areas not containing assets. This strategy aims at delaying the attacker, increasing the number of code areas that he or she has to analyze, and potentially making he or she lose time in the analysis of code not holding any value security-wise.

The *fingerprint enlargement* strategy is based on the extension of the scope of protections deployed on the asset protection phase, deploying them also on code areas adjacent to the assets. In this way, the attacker that finds the location of an asset has to analyze a higher amount of code, thus increasing the time needed to breach the asset security requirements. Finally, the *fingerprint shadowing* strategy conceals the fingerprints introduced in the code during the asset protection phase, deploying other protections on top the ones deployed on assets in the aforementioned phase.

For each asset hiding strategy, the effectiveness in hiding the fingerprints introduced by the protection techniques employed by ESP has been evaluated by the software security experts involved in the ASPIRE project, leading to the results reported in Table 6.1. The application of each strategy for each protection technique is rated HIGH when it can be very effective in hiding the fingerprint introduced by the technique, or LOW when applying the strategy for the protection may lead to minimal results.

For some protection techniques, not all the available strategies are appropriate. For instance, static remote attestation (see Section 1.1.3) is not suitable to shadow any kind of fingerprint, since it does not modify the code on which is deployed, but enforces checks on its integrity using ad-hoc external functions typically running on separate threads, as in the case of the implementation of this technique supported by ESP. Instead, different binary obfuscation techniques (see Section 1.1.2) may be effectively combined to obtain less noticeable fingerprints. An example is the insertion of opaque predicates in a code region previously protected with CFF, since this may alter the typical CFG resulting from this technique, depicted in Figure 6.1.

Indeed, there are protections for which the application of some asset hiding strategies may have the undesired effect of decreasing the global security level of the application. For example, this happens if virtualization obfuscations fingerprints (see Section 1.1.2) are replicated or enlarged throughout the code. Such fingerprints are evident for the attacker, as stated in the previous section, due to the custom VM bytecode introduced in the application binary, and the replication and enlargement techniques would be suitable to hide the location of the assets safeguarded by this technique. However, protecting a large amount of code with virtualizations obfuscation may reduce the effort needed by the attacker to reconstruct the mapping between the bytecode instructions and the assembler ones, since he or she would have more bytecode that can be analyzed for this purpose. Thus, the use of this technique in the asset hiding phase must be limited.

Also, another factor that must be taken into account, when assessing the suitability of a strategy for a protection technique, is the effect of the application of the strategy to the application performances. For example, the system calls introduced in protected code by anti-debugging techniques (see Section 1.1.4) may be easily recognizable by an attacker (e.g., using tools like

strace[3] in Linux). Thus, using the replication strategy to introduce additional debugging system calls can lead the attacker to analyze non-critical code. Conversely, anti-debugging has non-negligible overhead due to the debugger-debuggee context switches, thus replicating too much this technique fingerprints may lead to an unusable protected application.

Summarizing, the following factors must be taken into account when deciding the asset hiding additional protections:

- effectiveness of each strategy in hiding the protection fingerprints introduced in the asset protection phase;
- decreased efficacy of some protections techniques in securing the assets, when treated with specific asset hiding strategies;
- application performance overheads introduced by the asset hiding protections;
- forbidden precedences among protections, which, as in the asset protection phase (see Section 5.3), must not be introduced in the protection solution, since they would lead to a non-working protected application.

Indeed, computing a good asset hiding solution is not a trivial task, due to the high number of factors influencing the decision of additional protections used for this purpose.

6.3 Mixed Integer-Linear Programming model

This section presents the MILP problem that has been devised to model the process of deciding the additional protections that must be used to refine the solution inferred by ESP in the asset protection phase (see Chapter 5).

Mixed Integer-Linear Programming is a branch of operation research, encompassing optimization problems that contain variables constrained to integer values. While there are no previous applications of MILP problems to IT security in literature, decision-making processes in other fields of IT have been often modelled with this optimization technique, especially to tackle resource allocation problems. Indeed, the asset hiding decision process is a problem of this kind, since the overhead introduced by the additional protection constrains the generation of the asset hiding solution. For example, Metwally *et al.* use MILP problems in cloud IaaS scenarios to efficiently allocate resources in data centers[89]. Also, the book by Sankaralingam *et al.* on computer architectures [107] contains various applications of MILP problems in this field, for example the efficient generation of instruction sets for application-specific processors. Furthermore, various decision processes in the telecommunications field are solved using this optimization technique [21, 83].

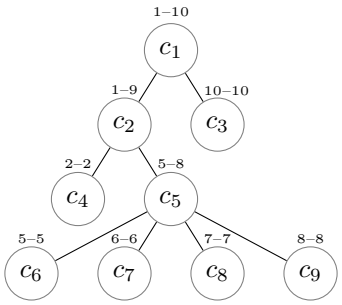
6.3.1 Application structure and protections

This section reports on how the data contained in the ESP Knowledge Base needed to drive the asset hiding phase, in particular the target application code structure and the protection techniques supported by ESP, is formalized in mathematically in order to employ it in the MILP problem. ESP can analyze code written in the C and C++ programming languages.

[3] https://www.systutorials.com/docs/linux/man/1-strace/

```
1   if (y == 7)
2   n = 1;
3   else
4   {
5   r = 7 + y;
6   x = FALSE;
7   y++;
8   n <<= 3;
9   }
10  n--;
```

(a) C source code.



(b) code region tree.

Figure 6.2: Code regions tree representing a C source code.

Code Regions

A code region, denoted in the with c, is a piece of code that is syntactically valid if parsed in isolation, and is individuated by specifying the source code file containing it, the starting and ending lines of the code comprised in the code region. The functions source(c), sl(c) and el(c) return respectively a reference to the source file (e.g., a path or an ID), the starting and ending line of a code region c. All the code regions that can be defined in the source code of the target application form the *code region space* C, while the ones that are taken into account to carry out the asset hiding phase form the *reduced code region space* $C^* \subseteq C$. All the code regions protected after the asset protection phase belong to C^*, since they constitute the starting point for the asset hiding phase.

A code region may contain another one, thus a *containment relation* is needed in the code region space. A code region c_i strictly contains c_j if $\text{source}(c_i) = \text{source}(c_j) \wedge \text{sl}(c_i) \leq \text{sl}(c_j) \wedge \text{el}(c_i) \geq \text{el}(c_j) \wedge (\text{sl}(c_i) < \text{sl}(c_j) \vee \text{el}(c_i) > \text{el}(c_j))$, i.e., if they are comprised in the same source file and c_i includes the lines of code constituting the c_j. Thus, this relation allows to represent the code region comprised in a source code file with a tree-like structure, with non overlapping siblings nodes, and ancestor nodes strictly containing their descendants. Thus, a function or class method is represented as a code region completely wrapping its body. Similarly, an application is a forest of code region trees, each of them representing one of the application source code files. An example of code region tree, obtained from a C snippet, is depicted in Fig. 6.2.

A partial order among code regions in the same tree can also be established. A code region c_i *precedes* (or *follows*) the code region c_j iff $\text{source}(c_i) = \text{source}(c_j) \wedge \text{parent}(c_i) = \text{parent}(c_j) \wedge \text{el}(c_i) < \text{sl}(c_j)$. These relationships are indicated with the notation $c_i \prec c_j$ (or $c_i \succ c_j$).

Also, a way to tell if two code regions are adjacent is needed, when deciding if the enlargement technique can be deployed, since two adjacent code regions can be merged and protected with the same pass. The function $\text{adj}(c_i, c_j) \in \{0, 1\}$ is used for this purpose:

$$\text{adj}(c_i, c_j) = \begin{cases} 1 & \text{if } (c_i \prec c_j) \wedge (\nexists c_k : c_i \prec c_k \prec c_j) \\ 0 & \text{otherwise} \end{cases}$$

Protections

Protections and their application to code regions must be formally modelled, since asset hiding strategies are implemented by applying protections to the code. First, the symbol P_i denotes a *protection* technique, while a protection instance (a protection with all the related configuration parameters, see Section 3.3) is represented with the symbol p_j; thus, $p_j \in P_i$. The function

protection$(p_j) = P_i$ returns the protection technique implemented by the protection instance p_j. The *protection instantiation space* $\mathbb{P}$ is the set of all the PIs contained in, which is $\mathbb{P} = \bigcup_i P_i$. The symbol $\mathbb{P}|_{\text{AP}}$ indicates the restriction of $\mathbb{P}$ to the PIs only used in the asset protection phase. Finally, the notation p/c indicates an Deployed Protection Instance (DPI), representing the use of a PI p to protect the code region c.

An *asset protection solution* is the output of the asset protection phase, and is defined formally as a poset of deployed PIs (see Chapter 5). The order of two deployed PIs p_i/c_j and p_k/c_l only matters if c_j and c_l share at least one line of code, i.e., $c_j = c_l$, $c_l \in$ ancestors(c_j) or $c_l \in$ descendants(c_j). Similarly, the *asset hiding solution* is the poset of deployed PIs generated by the asset hiding phase, refining the asset protection solution. To ease comprehension of the model, two helper functions are introduced. $\text{AP}(p/c) \in \{0, 1\}$ verifies if p/c is in the asset protection solution. $\text{AP}^*(p/c) \in \{0, 1\}$ checks if p is deployed to some code regions contained in c, i.e., code regions that are descendants of c.

$$\text{AP}(p/c) = \begin{cases} 1 & \text{if } p/c \text{ is in the asset protection solution} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{AP}^*(p/c) = \begin{cases} 1 & \text{if } \exists c_i \in \text{descendants}(c) : \text{AP}(p/c_i) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Also, a distinction must be made between horizontal and vertical enlargement. The first is enforced enlarging a DPI in the asset protection solution to contiguous code regions, e.g., referring to the code depicted to Fig. 6.2, if the DPI p_i/c_7 is enlarged to the code regions c_6, c_8 or c_9. Vertical enlargement instead happens when a DPI is enlarged to a code region that contains it, e.g., in Fig. 6.2, if p_i/c_7 is enlarged to c_5, c_2 or c_1.

Dependencies Among Protections

As already detailed in Section 5.3, some orderings among the protections in a protection solution are not valid, since applying protections using such orderings would lead to a non-working application. Thus, a partial ordering is needed among protections P_i. Formally, a protection P_i is a *forbidden precedence* of another protection P_j if p_i/c_k cannot be present before p_j/c_l in a valid solution, for all the intersecting code regions c_k and c_l. Also, a *singleton* protection (see Section 5.3) cannot be deployed more than once to the same code region. An example of this kind of protection is code mobility (see Section 1.1.3), since code can be moved to the trusted server only once.

Regarding the protection overheads, they are computed by ESP for all the code regions $c \in C^*$ during the generation of the model, using the formulas described in Section 5.4.3, and in the model are comprised in the set $\mathscr{O}_j$. Given a DPI p/c, $\mathscr{O}_j(p/c, \mathcal{M}_1(c), \mathcal{M}_2(c), \dots)$ indicates that $\mathscr{O}_j$ computation is based on the metrics $\mathcal{M}_1(c), \mathcal{M}_2(c), \dots$ computed on c. Since overheads are precomputed prior to model generation, and their dependency to metrics is not needed in the definition of the model, the compact notation $\mathscr{O}_j(p/c)$ will be used in the remainder of the chapter for the sake of readability.

6.3.2 Domain Parameters

The MILP model is built taking into account the reduced code region space $C^* = \{c_1, c_2, \dots\}$ and the PI set used in the asset protection solution $\mathbb{P}|_{\text{AP}} = \{p_1, p_2, \dots\}$.

The constant Γ specifies the maximum number of times a PI can be deployed on the same code region. The symbol Ω_i indicates the user-defined upper bound for the i-th overhead type.

The MILP model goal is maximizing the delay introduced by the asset hiding solution to tasks performed by the attacker. Thus, a way to estimate the delay introduced by every DPI is needed. The *confusion index* is introduced for this purpose. It is a parameter built to be proportional to the delay, but it is not intended to estimate the actual time needed by an attacker to find the application assets. In other words, while increasing the confusion index induces an increase in time needed by the attacker, other properties, e.g., a formula to translate it into a measure of attacker time, cannot be deduced. This represents a limitation of the current model. However, a mapping could be devised in the future, executing empirical assessments of the time needed by an attacker to find the assets in a protected application.

The set $\mathbb{T}$ contains the asset hiding strategies, i.e., $\mathbb{T} = \{\mathcal{R}, \mathcal{E}, \mathcal{S}\}$ where $\mathcal{R}$ is fingerprint replication, $\mathcal{E}$ is fingerprint enlargement, and $\mathcal{S}$ is fingerprint shadowing. A strategy-specific formula returns, for each DPI, the confusion index. The latter is dependent from the code region complexity metrics (see Section 5.4.1). The confusion indices associated to each deployed PI are used in the objective function as multiplicative factors, also called *bonuses* in the remainder of the chapter.

Strategy formulas can be unary, such as $\text{cnf}_{\mathcal{R},p_i}(\mathcal{M}_1(c), \mathcal{M}_2(c), \dots)$ and $\text{cnf}_{\mathcal{E},p_i}(\mathcal{M}_1(c), \mathcal{M}_2(c), \dots)$, which return the confusion index introduced by applying p_i/c to enforce the strategy $s \in \{\mathcal{R}, \mathcal{E}\}$. Alternatively, formulas may be binary. For example fingerprint shadowing is associated with this kind of formulas, where $\text{cnf}_{\mathcal{S},p_i,p_j}(\mathcal{M}_1(c), \mathcal{M}_2(c), \dots)$ returns the confusion index introduced by shadowing the p_j/c fingerprint, using the DPI p_i/c. $\text{cnf}_{\mathcal{S},p_i,p_j}(\mathcal{M}_1(c), \mathcal{M}_2(c), \dots)$ are bounded between 0 and 1: $\text{cnf}_{\mathcal{S},p_i,p_j}(\mathcal{M}_1(c), \mathcal{M}_2(c), \dots) = 1$, if applying p_i/c on top of p_j/c completely masks the fingerprint of p_j/c, and $\text{cnf}_{\mathcal{S},p_i,p_j}(\mathcal{M}_1(c), \mathcal{M}_2(c), \dots) = 0$ implies that applying p_i/c on top of p_j/c have not effect in masking the p_j/c fingerprint. Confusion indices are computed during the MILP model generation phase, thus the simplified notation $\text{cnf}_{\mathcal{R}}(p_i/c)$, $\text{cnf}_{\mathcal{E}}(p_i/c)$ and $\text{cnf}_{\mathcal{S}}(p_i/c, p_j/c)$ will be used when the reference to metrics of code region c is not needed.

6.3.3 Linear Problem

In this section, the Mixed Integer-Linear Programming (MILP) problem to decide the asset hiding protection is defined. Its goal is the maximization of the confusion index using the strategies in the set $\mathbb{T}$. The model is a specialization of the well-known Knapsack Problem (KP), using multiple capacity constraints to limit the introduced overhead. The examples provided in the model presentation are referred to the code region tree in Figure 6.2.

Variables

First, the MILP variables are introduced. The variable $x_{p_i/c_j} \in \{0, 1\}$ is set to 1 iff the PI p_i will be deployed either directly to the code region c_j, or to another that contains it, i.e., in $c_k \in \text{ancestors}(c_j)$. For instance, if $x_{p_i/c_2} = 1$, then the asset hiding solution either contains p_i/c_2, or p_i/c_1, since a protection deployed on a code region is implicitly deployed to all the code regions inside it.

The ordering between two PIs deployed to the same code region is modelled using the binary variable y_{c_i,p_j,p_k}, i.e., $y_{c_i,p_j,p_k} = 1$ iff p_j and p_k are both deployed on c_i and in this order. It is zero if the protections are deployed in the reverse order or if either $x_{p_j/c_i} = 0$ or $x_{p_k/c_i} = 0$. Clearly, it is not possible for two inverse variables y_{c_i,p_j,p_k} and y_{c_i,p_k,p_j} to be both equal to 1. Referring to the example code, given $x_{p_1/c_1} = 1$ and $x_{p_2/c_1} = 1$, if the $y_{c_1,p_1,p_2} = 1$ then p_1 will be deployed to c_1 before p_2, and y_{c_1,p_1,p_2} will be forced to 0.

This behavior is imposed introducing the following constraints, i.e., if both x_{p_j/c_i} and x_{p_k/c_i} are equal to 1, only one of the two variables y_{c_i,p_j,p_k} and y_{c_i,p_k,p_j} must be equal to 1, otherwise

both y_{c_i,p_j,p_k} and y_{c_i,p_k,p_j} must be equal to 0:

$$0 \leq -2y_{c_i,p_j,p_k} - 2y_{c_i,p_k,p_j} + x_{p_j/c_i} + x_{p_k/c_i} \leq 1, \forall i,j,k$$

A second set of constraints avoid the insertion of cycles in the ordering of protections deployed to the same code region (for more than three protections). In other words, loops like $y_{c_i,p_j,p_k} = y_{c_i,p_k,p_l} = y_{c_i,p_j,p_l} = 1$ are avoided. This is obtained with the following constraints:

$$y_{c_i,p_j,p_k} - y_{c_i,p_k,p_l} - y_{c_i,p_j,p_l} \leq -1, \forall i,j,k,l : j \neq k \neq l$$

Also, the model forces $y_{c_i,p_j,p_k} = 0$ if p_j is a forbidden precedence for p_k.

The binary variables g_{c_i,c_j,p_k}, which are defined only for adjacent code regions c_i and c_j, are used to decide if two code regions must be merged when deploying protections (horizontal enlargement). If $g_{c_i,c_j,p_k} = 0$ and $x_{p_k/c_i} = x_{p_k/c_j} = 1$, two separate DPIs p_k/c_i and p_k/c_j will be comprised in the asset hiding solution. If $g_{c_i,c_j,p_k} = 1$, the asset hiding solution will contain a DPI p_k/c_l, where c_l comprises the union of code regions c_i and c_j. In the example, if p_1/c_7 is in the asset protection solution and $g_{c_6,c_7,p_1} = 1$, then the asset hiding solution will contain p_1/c_{10}, where c_{10} is a code region $c_6 \cup c_7$. If, otherwise, $g_{c_6,c_7,p_1} = 0$, then the asset hiding solution will contain p_1/c_6 and p_1/c_7.

Furthermore, $g_{c_i,c_j,p_k} = 1$ implies $x_{p_k/c_i} = x_{p_k/c_j} = 1$, obtained imposing the following set of constraints:

$$2g_{c_i,c_j,p_k} - x_{p_k/c_i} - x_{p_k/c_j} \leq 0, \quad \forall i,j,k : \mathrm{adj}(c_i,c_j) = 1$$

Some other variables are needed to correctly process the bonuses used in the objective function. Replications must be avoided, i.e., since the same operation can be interpreted as a consequence of more strategies, e.g., enlargement and shadowing. Furthermore, containment relations must be managed, so that while $x_{p_i/c_j} = 1$ forces $x_{p_i/c_k} = 1$ for all its descendants c_k, the bonus is considered only for the larger code region.

Thus, the additional binary variables $e_{c_i,c_j,p_k} \in \{0,1\}$ are introduced, representing enlargements spanning both horizontally and vertically to increase the code that must be protected. To compute the bonuses of descendants in a proper way, the e_{c_i,c_j,p_k} keep track of protections deployed due to enlargements. An example can be made considering a deployed PI p_1/c_7 in the asset protection solution, which, during the asset hiding phase, is first vertically enlarged to c_5; then, c_5 is enlarged to c_4. In this case $e_{c_4,c_5,p_1} = 1$, since p_1 has been deployed to c_4 to enlarge the asset protection DPI p_1/c_7, and c_4 is a sibling of c_5, which contains the originally protected code region c_7. Generalizing, the variable e_{c_i,c_j,p_k} set to 1 means that the asset hiding solution contains a PI p_k deployed to a code region containing both c_i and c_j, due to an enlargement of the PI p_k that in the asset protection solution was originally deployed either to c_j or one of its descendants.

The e_{c_i,c_j,p_k} summarizes complex enlargements, thus they can be obtained as composition of the g_{c_i,c_j,p_k}, which define simple enlargements:

$$e_{c_i,c_j,p_k} = \bigwedge_{g_{c_l,c_m,p_k} \in G_{c_i,c_j,p_k}} g_{c_l,c_m,p_k} \tag{6.1}$$

where G_{c_i,c_j,p_k} reports the individual $g_{c,c,p}$ affected by the complex enlargement:

$$G_{c_i,c_j,p_k} = \{\ g_{c_l,c_m,p_k} \ : \ \mathrm{AP}^*(p_k/c_j) \wedge (c_i \preceq c_l \prec c_m \preceq c_j \vee c_j \preceq c_l \prec c_m \preceq c_i)\ \} \tag{6.2}$$

For example, if p_1/c_6 is in the asset protection solution and p_1/c_8 is the consequence of two enlargements, first from c_6 to c_7 ($g_{c_6,c_7,p_1} = 1$) and then from c_7 to c_8 ($g_{c_7,c_8,p_1} = 1$), this leads to $G_{c_6,c_8,p_1} = \{g_{c_6,c_7,p_1}, g_{c_7,c_8,p_1}\}$ and $e_{c_6,c_8,p_1} = g_{c_6,c_7,p_1} \wedge g_{c_7,c_8,p_1} = 1$.

It is also necessary to identify if a DPI derives from an enlargement or replication, in order to compute the bonuses properly, because these two techniques are mutually exclusive. To this purpose, the auxiliary binary variables z_{p_i/c_j} are introduced in order to identify which $x_{p/c}$ are set to 1 in the asset hiding model solution due to fingerprint enlargements. We have $z_{p_i/c_j} = 1$ when p_i has been deployed to c_j for enlarging an DPI in the asset protection solution or 0 if another technique has been used, which is

$$z_{p_i/c_j} = \bigvee_{e_{c_j,c_k,p_i} \in E_{p_i/c_j}} e_{c_j,c_k,p_l} \tag{6.3}$$

where E_{p_i/c_j} includes the variables that may indicate if PIs deployed to the siblings c_k of c_j are consequence of complex enlargements.

$$E_{p_i/c_j} = \{\ e_{c_j,c_k,p_i} \ :\ \mathrm{AP}^*(p_i/c_k) = 1 \wedge \mathrm{parent}(c_j) = \mathrm{parent}(c_k)\ \}$$

Fingerprint shadowing and replication (or enlargement) are not mutually exclusive, thus when enlarging or replicating a DPI a fingerprint of another DPI may be shadowed. In this case, replicating the bonuses cannot happen, thus is not needed to this case not need to manage this case explicitly.

Objective Function

The following objective function is used by the MILP model, aiming to maximize the confusion index:

$$\begin{aligned} o.f.\ \max \Bigg(& \sum_{c_i \in C^*} \sum_{p_j \in \mathbb{P}|_{\mathrm{AP}}} \rho^{\mathcal{R}}_{p_j/c_i} (x_{p_j/c_i} - z_{p_j/c_i}) + \\ & \sum_{c_i \in C^*} \sum_{p_j \in \mathbb{P}|_{\mathrm{AP}}} \rho^{\mathcal{E}}_{p_j/c_i} \mathrm{AP}^*(p_j/c_i) x_{p_j/c_i} + \\ & \sum_{c_i \in C^*} \sum_{p_j \in \mathbb{P}|_{\mathrm{AP}}} \sum_{c_k \in C^*} \rho^{\mathcal{E}}_{p_j/c_i} e_{c_i,c_k,p_j} + \\ & \sum_{c_i \in C^*} \sum_{p_j \in \mathbb{P}} \rho^{\mathcal{S}}_{p_j/c_i} \Bigg) \end{aligned}$$

The first summation takes into account the replication strategy and it uses the $\rho^{\mathcal{R}}_{p_j/c_i}$ factor. As anticipated, the z_{p_j/c_i} variables distinguish if enlargement or replication have been deployed. Subtracting the z_{p_j/c_i} ensures that replication contributions are properly computed. The bonus is computed recursively on each code region tree:

$$\rho^{\mathcal{R}}_{p_j/c_i} = \mathrm{cnf}_{\mathcal{R}}(p_j/c_i) - \sum_{c_k \in \mathrm{descendants}(c_i)} \rho^{\mathcal{R}}_{p_j/c_k}$$

It should be noted that confusion indices of all the descendant code regions are subtracted to avoid duplications, since the summation in the objective function will consider them because it spans over all the code regions.

The second and third summations respectively report the contribution of the vertical and horizontal enlargement. The enlargement bonus $\rho^{\mathcal{E}}_{p_j/c_i}$ must ensure that only the deployed PIs that are consequence of some enlargement of deployed PIs in the asset protection solution are

considered. To this purpose, the vertical enlargement uses the $\mathrm{AP}^*(p_j/c_i)$ as an explicit multiplying factor. The horizontal enlargement uses the e_{c_i,c_k,p_j} variables, which have been defined ad hoc. As before, the enlargement bonus is computed recursively and the confusion indices are subtracted:

$$\rho^{\mathcal{E}}_{p_j/c_i} = \mathrm{cnf}_{\mathcal{E}}(p_j/c_i) - \sum_{c_k \in \mathrm{descendants}(c_i)} \rho^{\mathcal{E}}_{p_j/c_k}$$

Finally, the fourth summation takes into account the fingerprint shadowing, and similarly to the other summations use a shadowing bonus $\rho^{\mathcal{S}}_{p_j/c_i}$. The $\rho^{\mathcal{S}}_{p_j/c_i}$ bonus indicates how well the p_j/c_i fingerprint is masked by the protections deployed on top of it. As for $\mathrm{cnf}_{\mathcal{S}}(p_k/c_i, p_j/c_i)$, the $\rho^{\mathcal{S}}_{p_j/c_i}$ bonuses are in [0,1], where $\rho^{\mathcal{S}}_{p_j/c_i} = 1$ means that the fingerprint of p_j/c_i is completely masked and $\rho^{\mathcal{S}}_{p_j/c_i} = 0$ means no shadowing at all. Since the value of $\rho^{\mathcal{S}}_{p_j/c_i}$ is between 0 and 1, it is ensured that when a fingerprint has been already completely masked, no additional bonus is given if protections are added.

The bonus $\rho^{\mathcal{S}}_{p_j/c_i}$ is evaluated by summing up the $\mathrm{cnf}_{\mathcal{S}}(p_k/c_i, p_j/c_i)$ of all the p_k/c_i deployed to shadow p_j/c_i, which are recognized because $y_{c_i,p_j,p_k} = 1$. If the summation exceeds 1, the bonus $\rho^{\mathcal{S}}_{p_j/c_i}$ is set to 1.

Since $\rho^{\mathcal{S}}_{p_j/c_i}$ increases the objective function value, its evaluation is obtained by introducing the following set of constraints:

$$\begin{array}{rcll} 0 & \leq & \rho^{\mathcal{S}}_{p_j/c_i} & \leq & 1 \\ & & \rho^{\mathcal{S}}_{p_j/c_i} & \leq & \sum_{p_k \in \mathbb{P}} \mathrm{cnf}_{\mathcal{S}}(p_k/c_i, p_j/c_i) \cdot y_{c_i,p_j,p_k} \end{array}$$

Simply increasing the bonuses, assets and strategies may be prioritized, e.g. by weighting the objective function summations or by using multiplicative factors that take into account the importance of the asset.

Finally, the e_{c_i,c_j,p_k} (defined in Equation 6.1) and z_{p_i/c_j} (defined in Equation 6.3) are linearized as follows:

$$\begin{cases} e_{c_i,c_j,p_k} \leq \dfrac{\sum_{g_{c_l,c_m,p_k} \in G_{c_i,c_j,p_k}} g_{c_l,c_m,p_k}}{\left|G_{c_i,c_j,p_k}\right|} & \text{if } \left|G_{c_i,c_j,p_k}\right| > 0 \\ e_{c_i,c_j,p_k} = 0 & \text{otherwise} \end{cases}$$

$$\begin{cases} z_{p_i/c_j} \geq \dfrac{\sum_{e_{c_l,c_m,p_i} \in E_{p_i/c_j}} e_{c_l,c_m,p_i}}{\left|E_{p_i/c_j}\right|} & \text{if } \left|E_{p_i/c_j}\right| > 0 \\ z_{p_i/c_j} = 0 & \text{otherwise} \end{cases}$$

Additional Constraints

As described in Chapter 1, the asset hiding solution must not exceed the user-defined overheads. Therefore a set of inequalities are introduced, constituting the KP-problem capacity constraints:

$$\sum_{c_j \in C^*} \sum_{p_i \in \mathbb{P}|_{\mathrm{AP}}} x_{p_i/c_j}\, \omega_o(p_i/c_j) \leq \Omega_o, \quad \forall o$$

where $\omega_o(p_i/c_j)$ is a constant defined as:

$$\omega_o(p_i/c_j) = \mathscr{O}_o(p_i/c_j) - \sum_{c_k \in \mathrm{descendants}(c_j)} \omega_o(p_i/c_k)$$

It should be noted that also in this case, the code regions' overheads added by the descendants must be subtracted, since they would be counted more than once during the double summation.

Containment among code regions may also have consequences on the order and applicability of protections. First, if the set of protection deployed to a code region is different from the ones deployed to its parent, then the deployed protections associated with the child but not to the parent must be deployed before the protections that are deployed to both. Second, if a set of PIs is deployed to a code region, the same set of PIs must be deployed to all its descendant code regions and in the same order.

Consequently, the MILP model contains three additional sets of constraints, which use the following set of variables that regulate the possibility to merge adjacent code regions:

$$\Psi_{p_i/c_j} = \{\ g_{c_k,c_l,p_i}\ :\ \text{parent}(c_k) = \text{parent}(c_l) = c_j\ \}$$

The first set of constraints ensures that deployed a PI to a code region implies that it must be deployed to all the merged children code regions, and formally, they are the AND of variables in Ψ_{p_i/c_j}:

$$\left|\Psi_{p_i/c_j}\right| x_{p_i/c_j} - \sum_{g_{c_k,c_l,p_i} \in \Psi_{p_i/c_j}} g_{c_k,c_l,p_i} \leq 0, \forall i,j,k,l$$

That is, $x_{p_i/c_j} = 1$ forces the $g_{c_k,c_l,p_j} = 1$ for all the adjacent children c_k and c_l of c_j.

The second set of constraints guarantees that if two PIs are deployed to a code region, they are deployed in the same order to all the children code regions, which is:

$$y_{c_i,p_j,p_k} \leq y_{c_l,p_j,p_k}, \forall i,j,k,l : \text{parent}(c_l) = c_i$$

Finally, the third set ensures that, when merging adjacent code regions PIs must be deployed in the same order, which is:

$$\begin{aligned}&\big(\big(g_{c_i,c_j,p_k} = 1\big) \wedge \big(y_{c_i,p_k,p_l} = 1\big)\big) \rightarrow \\ &\big(\big(g_{c_i,c_j,p_l} = 1\big) \wedge \big(y_{c_j,p_k,p_l} = 1\big)\big), \forall i,j,k,l\end{aligned}$$

For the sake of readability, the last constraints contains the traditional logical operators $\wedge$ and $\rightarrow$. Such expressions can be easily linearized via standard techniques.

6.4 Translation algorithm

The *translation algorithm* generates the final asset hiding solution, which is, the ordered list of protections to apply on every code region. It takes as input the values assumed by the x_{p_i/c_j}, y_{c_i,p_j,p_k} and g_{c_i,c_j,p_k} variables, whose values have been obtained by solving the MILP problem presented in Section 3. These variables express the protections that must be deployed on every code region, the order of deployment, and if the protections need to be deployed on adjacent code region as a whole or as individual applications of the same protection. For instance, variables corresponding to protections that are enabled on ancestors need to be enabled for all the descendants as well. However, in practice, they must be deployed only once on the ancestor.

Moreover, code regions that are non-overlapping (sibling of the same ancestors, which may be adjacent or not) need to be merged according to the g_{c_i,c_j,p_k} variables.

In order to globally determine the protections that must be deployed, the translation algorithm has been developed to perform:

- *generate lists*, every code region is associated with a set of protections x_{p_i/c_j} that needs to be deployed on it. However, protections must be deployed according to the order given by the y_{c_i,p_j,p_k} variables. The algorithm transforms the information in these variables into a list of deployed PIs associated to each node.

- *remove duplicate*, if a protection is deployed to an ancestor, the protection is removed from the list of PIs deployed to each of its descendants.
- *horizontal enlargement*, if a protection needs to be deployed to two or more adjacent code regions (because of the x_{p_i/c_j}) and can be merged (due to g_{c_i,c_j,p_k} values), a new code region is generated that spans all the adjacent code regions to merge. This node becomes an intermediate ancestor, i.e., a child of the original parent and the parent of all the merged node.

After these operations are completed, the translation algorithm outputs the asset hiding solution, i.e., a coherent set of lists of protections to apply on every code region, i.e., a poset of PIs.

6.5 Validation

In this section, the asset hiding phase described in this chapter is compared with ESP requisites listed in Section 2.1.

Requisites for the ESP usage scenario, defined in Section 2.1.1, are mostly respected by the asset hiding phase. First, the time needed to execute this phase can be influenced with a set of configuration parameters: a hard time limit, and a maximum number of non-asset code regions taken into account when generating the MILP model. A longer execution of the algorithm may be adopted prior to the target application release, in order to find a better hiding solution, while shorter times are suitable when patches are released and time is an issue. Thus, the asset hiding phase can adapt to the target application life-cycle. Also, asset hiding solutions are human-readable, being presented with the same format of asset protection ones; furthermore, for each DPI added in the solution, ESP presents the user with the strategy (or strategies) for which the DPI has been chosen.

A limitation of the current approach is that the attacker skills are not taken into account when the additional protections are decided. This is due to the lack of experimental assessments of the asset identification capabilities of the attackers, which the authors intend to execute in the future. After this assessments, the data needed to take into account the attacker profile in the asset hiding decision process would be obtained, and furthermore, a method to obtain an actual estimation of the time needed to find the assets could be devised.

Regarding the requirements for the risk mitigation phase (Section 2.1.3), ESP respects them. First, since the identification of the assets is a necessary preliminary step for attacks, the asset hiding phase defers the attacks, hiding the fingerprints of the asset protection phase that ease the detection assets by attackers. Furthermore, the asset hiding protections are chosen to maximize the confusion induced by them in the attacker, which is evaluated through formulas that are dependent on the metrics of the code regions on which the formulas are deployed. Overheads are taken into account in this phase, inferring solutions that, when deployed, lead to protected binaries that are still usable. Finally, as the asset protection phase, all the protection techniques used in this phase are implemented by automatic protection tools, so also this phase is completely automated.

Finally, in the ASPIRE project validation phase [13] the asset hiding phase was deemed as a useful addition to the ESP workflow. The experts validated the additional protections inferred in this phase, stating that the proposed solutions for the ESP use cases were free from inconsistencies, and would be useful to defer an attacker in the process of finding the protected assets in the target application binary. Furthermore, they considered it an advancement in respect with the current way of handling fingerprints by software security practitioners, which is essentially based on various obfuscation attempts on the application binary, a time-consuming process that is needed to find a good trade-off between asset concealment and introduced overhead.

Chapter 7

Conclusions and future work

> No book can ever be finished. While working on it we learn just enough to find it immature the moment we turn away from it.
>
> Karl Popper

This thesis has presented ESP, an expert system able to automatically protect the assets comprised in an application code. With the implemented workflow, it mimics the mental processes of a software security expert, tasked with the protection of an application. Given the source code of a program that must be protected, ESP is able to produce a binary program, protected with a comprehensive set of software security techniques against the possible attacks that can be mounted to endanger the application assets. Due to its high degree of automation, requiring from the user only the definition of a set of security requirements for the application assets that must be safeguarded from possible threats, it can be easily adopted by a developer, without any prior experience in software security, which wants to protect its application. Furthermore, its results can be leveraged by security experts, which can analyze the inferred attacks against the application, and the protection techniques deemed suitable to defer such attacks, to evaluate the proposed security solution, and manually refine it, if not completely satisfied with the proposed results.

ESP development resulted in various advancement of the state of the art in the field of software security, described in the following. First, a comprehensive meta-model (Chapter 3) for software security has been defined, able to formalize all the concepts involved in the application protection process, including the characteristics of the components of the application code that must be protected, the assets and their security requirements, the possible attacks that can be mounted against them, and the protection techniques that can be deployed on the application code to defer such assets. Then, a comprehensive set of software risk management methodologies have been presented, organized in a complete workflow for securing applications (Chapter 2). The workflow starts with a risk assessment phase (Chapter 4), inferring, from a set of abstract simple attacker tasks, complete attack graphs modelling the possible courses of actions that an attacker may carry out to breach the assets security requirements. Then, a risk mitigation phase (Chapter 5) decides, among all the available protection techniques, the best combination of them that can be deployed on each asset code to defer, for as long as possible, the successful execution of the attacks inferred in the previous phase. In doing so, this phase mimics not only the protection decision process of the security expert, but also, using a game-theoretic approach, the subsequent assessment of the effectiveness of such protections, which is typically undertook in software security companies by

skilled white-hat hackers, trying the resistance of the protected application against attacks on the field. Finally, the concept of protection fingerprint is introduced (Chapter 6), i.e., peculiar forms taken by code or run-time behaviours assumed by the application after deployment of protections, which can be leveraged by an attacker to identify the valuable assets inside the application binary. The risk management methodology responds to this problem with an additional asset hiding phase, which, with the deployment of protections on areas of code that are not security-sensitive, is able to confuse the attacker, which can be lead to concentrate its attacks on code that has no value for him. Finally, leveraging a set of automatic protection tools (Section 1.1.1), ESP is able to actually apply the inferred best protection solution to the application source code, building a protected program binary, thus completely automating the software protection process.

Software security experts have assessed the quality of results produced by ESP: while judging it not ready for actual use by software security companies, they deemed the system promising. Indeed, the author intends to further improve the devised methodologies for application risk management, with the final objective of making ESP a professional tool that can be used by experts to simplify their work. First, more attack steps need to be modelled, in order to infer attack paths that better approximate the attacker courses of action. Also, due to the recent introduction of frameworks able to automatize the execution of attacks on application, for example angr[1], the mitigation phase can be further improved by actually testing the protected binary with attacks carried out with this tool, thus better assessing the effectiveness of protections in deferring such attacks. Finally, a comprehensive set of empirical assessments of the results produced by ESP must be undertaken: the author has already participated in experiments devised to assess the effectiveness of specific software protections [123], and intends to devise new experiments to better tune the ESP decision processes.

Concluding, the presented software security workflow, and the included risk management methodologies, can be a starting point towards the complete automation of software protection processes, with results comparable to the manual protection of applications by experts in the field, but also reducing the risk of vulnerabilities introduced by errors, always looming when humans are engaged in the execution of such complex processes.

[1] https://github.com/angr

Appendix A

ESP implementation

This appendix reports the main information about the actual implementation of the ESP, whose source code is released with an Eclipse Public License 1.0[1], and is available on a GitHub repository[2].

A.1 Main ESP components

The ESP is implemented as a set of Eclipse Platform[3] plug-ins, written in the Java programming language. The ESP code is organized in the following Java packages[4]:

- `esp`: the ESP main package, coordinating other packages methods to implement the workflow described in Section 2.2.5;
- `esp.attacks`: implements the Risk Assessment Engine reasoning process, described in Chapter 4;
- `esp.connector`: contains methods for connecting the ESP with external tools, presented later in this section, needed by the Source Code Analyzer, Protection Tools Connector and Solution Deployer;
- `esp.l1p`: implements the Risk Mitigation Engine, described in Chapter 5;
- `esp.l2p`: comprises the code for Asset Hiding Engine reasoning process, outlined in Chapter 6;
- `esp.metrics`: contains the Metrics Framework code, used to evaluate complexity metrics on the application binary, with the methodologies described in Section 5.4;
- `esp.optimizationAPI`: implements a compatibility layer to abstract the APIs of external MILP solvers, needed by the Asset Hiding Engine, supporting at time of writing IBM ILOG CPlex Optimizer[5] and lp_solve[6];

[1] https://www.eclipse.org/legal/epl-v10.html

[2] https://github.com/daniele-canavese/esp

[3] https://www.eclipse.org/eclipse/eclipse-charter.php

[4] All packages are comprised in the `it.polito.security` root package, which is not included in the following package names for the sake of readability.

[5] https://www.ibm.com/it-it/analytics/cplex-optimizer

[6] http://lpsolve.sourceforge.net/

- esp.protections: contains the Protection Enumerator code, detailed in Section 5.2;
- esp.kb: implements the Knowledge Base, with the meta-model a-priori information (see Chapter 3) gathered from the software security experts during the ASPIRE project;
- esp.ui: implements the User Interface, based on the Eclipse Platform UI[7];
- esp.util: contains a set of helper methods shared among the other plug-ins.

A.2 ESP workflow

The ESP workflow, used to generate the application protected binary starting from the application source code, can be subdivided in the following main phases:

1. source code parsing (Section A.1): the Source Code Analyzer produces a description of the code structure (e.g., functions, variables, CFG), while the Metrics Framework obtains the software metrics, both program-wide and function-wide; the result is adding to Knowledge Base an instance of the core application meta-model described in Section 3.2;
2. vulnerability analysis (Chapter 4): the Risk Assessment Engine, for each security requirement of each asset, infers the possible attacks that can endanger the requirement; such attacks are saved in the Knowledge Base, thus instancing the attack meta-model depicted in Section 3.4;
3. protections enumeration (Section 5.2): the Protection Enumerator selects the protections for each asset that can be useful to delay the attacks found during the vulnerability analysis, adding to the Knowledge Base an instance of the protection meta-model detailed in 3.3;
4. protections decision (Chapter 5): the Risk Mitigation Engine, given the suitable protections selected by the Protection Enumerator, finds the combination of protections, also called *protection solution*, which protects the application from the attacks found in the vulnerability analysis phase; the Risk Mitigation Engine assess the *score* of a solution, i.e., the efficacy of the solution in protecting the application, in function of the metrics evaluated on the binary protected with the solution; as described in Section 5.4, the protected binary metrics are predicted by the Metrics Framework for each considered solution, since deploying with the Solution Deployer all the solutions generated by the Risk Mitigation Engine would take an unfeasible time; this phase enriches the protection meta-model instance generated in the protections enumeration phase, adding the evaluated solutions with their score;
5. asset hiding (Chapter 6): the Asset Hiding Engine, given a solution (e.g., the one with the highest score), can enrich it with additional protections applied not only on assets, but also on other functions or snippets, in order to slow down the identification of assets by the attacker; the Asset Hiding Engine decides the additional protection generating a MILP optimization problem [19], feeding it to an external solver (e.g., IBM ILOG CPLEX Optimizer[8], whose result will be translated by the Asset Hiding Engine in a new solution in the Knowledge Base with the asset hiding additional protections; this phase is optional, and the ESP can be configured to skip it;

[7] https://www.eclipse.org/eclipse/platform-ui/

[8] https://www.ibm.com/it-it/analytics/cplex-optimizer

6. solution deployment (Section A.3): the Solution Deployer, given the solution selected by the user (with or without the asset hiding additional protections), executes the external protection tools configured accordingly to enforce the protections constituting the selected solution.

Following this workflow, the ESP is able to generate an application binary, (hopefully) protected against known attacks, basing its decision both on expert knowledge and on the specific characteristics of the application source code, with minimal user interaction: in the end, it mimics the mental processes of a software protection expert tasked to protect an application.

In the various phases of its workflow, the ESP relies on the following external tools and libraries:

- the ACTC, detailed in Section 1.1.1, called by the Metrics Framework to obtain metrics on the target application code, and by the Solution Deployer to deploy the chosen solution in order to obtain the protected binary;
- the Eclipse CDT, which parses the target application source code and produces an AST [79] representing its structure; this AST is used by the Source Code Analyzer to automatically instantiate the application meta-model described in Section 3.2;
- Swi-Prolog[9], an implementation of the Prolog language, used by the Risk Assessment Engine to execute the rules, written in the aforementioned language, which build the AP against the assets in the target application, as reported in Chapter 4;
- either IBM ILOG CPlex Optimizer or lp_solve, which are used by the Asset Hiding Engine to solve a MILP problem: the latter is defined by the Asset Hiding Engine to decide the additional protections that must be deployed to the target application in order to hide its protected assets from the attacker, as detailed in Chapter 6.

The ESP has been designed with a modular approach, in order to be easily extensible. New features can be added to the ESP implementing them as Eclipse Platform Plug-ins, and including them in the Manifest[10] file of the main package. Also, the external tools can be easily substituted with other equivalents, since the communication between each tool and the ESP is governed by an ad-hoc connector class, comprised in the Protection Tools Connector plug-in. For example, Eclipse CDT may be substituted with another C code parsing tool (e.g., a commercial one such as GrammaTech CodeSonar[11]) by simply implementing the connector class and substituting it to the CDT one. Also, to solve the MILP problem generated by the Asset Hiding Engine, any optimization solver can be used, by simply writing a class that implements a generic interface that abstracts optimizers APIs.

A.3 Solution deployment

This section describes how an ESP solution, built by the Risk Mitigation Engine (and optionally the Asset Hiding Engine), can be deployed by the Solution Deployer, ultimately obtaining a protected binary from the target application source code.

[9] http://www.swi-prolog.org/features.html

[10] https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html

[11] https://www.grammatech.com/products/source-code-analysis

The protections supported by the ESP can be applied to the target code calling the automated software protection tools described in Section 1.1.1 via the Protection Tools Connector. In particular, protections developed during the ASPIRE project can be deployed using the ACTC, while a set of source-to-source code and variable obfuscation techniques are deployed using Tigress.

First, the Solution Deployer gathers all the DPI that must be applied to the source code using Tigress, which is called accordingly, and produces the protected source code files: by comparing these with the original ones, a GNU patch[12] file is produced.

Then, the Solution Deployer gathers the DPI related to ASPIRE protection techniques, and adds to the patch file the annotations that drive the ACTC. It should be noted that the ESP supports a similar set of annotations as a comfortable way for the target application software developer to mark the assets and to specify their security requirements (see Section 3.2); in the following example, the code comprised between the `pragma` directives has been marked by its developer as an asset whose confidentiality must be preserved:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  char pwd[] = "hardcodedPassword";
5
6  int main()
7  {
8      char temp[20] = "";
9
10     _Pragma("ASPIRE begin requirement("confidentiality"));
11     printf("Insert password: ");
12
13     scanf("%20s",temp);
14
15     if(strcmp(temp,pwd)==0)
16         printf("Correct password!\n");
17     else
18         printf("Wrong password!\n");
19     _Pragma("ASPIRE end");
20
21     return 0;
22 }
```

In this case, the Solution Deployer will replace such annotations in the patch file with the ones, described in Section 1.1.1, used to drive the ACTC, specifying the actual protections that the ESP has decided to apply in order to preserve the abstract asset security requirement specified by the user. Also, the developer may explicitly ask the ESP to protect a code region with a specific protection, manually writing the related ACTC annotation in the code: in this case, the ESP will respect the user will, inserting the manually defined protection in all the solutions inferred by the Risk Mitigation Engine and Asset Hiding Engine.

[12] http://savannah.gnu.org/projects/patch/

Acronyms

ACCL ASPIRE Client-side Communication Logic. 22, 24, 25

ACTC ASPIRE Compiler Tool Chain. 8, 22, 42, 71, 74, 77–80, 110, 111

AI Artificial Intelligence. 80

ANN Artificial Neural Network. 81

AP Attack Path. 49, 71–73, 75, 84, 85, 87–90, 110

API Application Programming Interface. 9, 71, 95, 108, 110

AS Attack Step. 87–89

ASCL ASPIRE Server-side Communication Logic. 22, 24, 25, 83

ASPIRE Advanced Software Protection: Integration, Research and Exploitation. vi, 4, 5, 8, 9, 22, 24, 25, 27, 33, 39, 41–44, 48, 49, 52, 59, 67, 68, 74, 75, 80, 83, 88, 89, 91, 95, 96, 105, 109, 111, 112

AST Abstract Syntax Tree. 40, 110

AudES Expert System for security Auditing. 31

BSA BSA | The Software Alliance. 1, 2

BYOD Bring Your Own Device. 1

CAMEL Cloud Application Modelling & Execution Language. 60

CC Cyclomatic Complexity. vii, 78, 79, 82, 87, 88

CDT C/C++ Development Tooling. 40, 71, 76, 110

CFF Control Flow Flattening. 7, 8, 11, 12, 38, 69, 70, 87, 94, 96

CFG Control Flow Graph. 11, 12, 18, 36, 38, 52, 69, 70, 78, 79, 82, 86–88, 94–96, 109

CIL C Intermediate Language. 9

CPU Central Processing Unit. 4, 8, 9, 17, 42, 77, 79, 83, 94

CVE Common Vulnerabilities and Exposures. 39, 50

CWE Common Weakness Enumeration. 39, 50

CySeMoL Cyber Security Modeling Language. 60

DDoS Distributed Denial of Service. 22, 35

DEC Digital Equipment Corporation. 30, 31

DENDRAL DENDritic ALgorithm. 30

DIABLO Diablo Is A Better Link-time Optimizer. 8, 9, 11, 14–16, 23, 27, 37, 42, 43, 74, 77–82

DLL Dynamic-Link Library. 23

DPI Deployed Protection Instance. 42, 43, 47, 54–56, 71, 72, 74–76, 81, 89, 99–102, 105, 111

DRM Digital Rights Management. 49

ESP Expert system for Software Protection. vi, vii, 3–11, 17, 19, 22, 24, 27, 33–64, 66–69, 71–80, 83, 91, 93–97, 99, 105–111

FOSS Free and Open Source Software. 21, 26, 81

GCC GNU C Compiler. 40, 81

GDB GNU DeBugger. 26

GNU GNU is Not Unix. 40, 50, 111, 113

GUI Graphical User Interface. 37

HL Halstead's Length. vii, 78, 79, 82, 87

IDES Intrusion Detection Expert System. 31

IDS Intrusion Detection System. 31

IP Intellectual Property. 2

LISP LISt Processor. 30

MATE man-at-the-end. 3, 26, 34–36, 38, 39, 41, 43, 53, 84

MILP Mixed Integer-Linear Programming. 44, 93, 94, 97, 100, 102, 104, 105, 108–110

MITM man-in-the-middle. 22

ML Machine Learning. 80, 81

NADIR Network Anomaly Detection and Intrusion Reporter. 31

NIDES Next-generation Intrusion Detection Expert System. 31

NIDX Network Intrusion Detection eXpert system. 31

NIST National Institute of Standards and Technology. 34, 35, 68

OS Operating System. 4, 8, 23, 24, 26, 95

OTP One-Time Password. 66

OWL Web Ontology Language. 28, 114

OWL2 Web Ontology Language 2. 28, 40

PI Protection Instance. vii, 42, 50, 51, 54, 56, 72, 74, 75, 77, 78, 80, 81, 99–102, 104, 105

PO Protection Objective. 72, 74–77, 84

RAT Remote Access Trojan. 2, 21

SGX Software Guard Extensions. 24

SLOC Source Lines Of Code. vi, 49, 50

SMB Server Message Block. 10

SOM Self-Organizing Maps. 31

TCG Trusted Computing Group. 24

TPM Trusted Platform Module. 24

TXT Trusted eXecution Technology. 24

VAX Virtual Address eXtension. 30

XCON eXpert CONfigurer. 30, 31

XML eXtensible Mark-up Language. 74

Bibliography

[1] 105th United States Congress. "Digital Millennium Copyright Act (Pub. L. No. 105-304)". In: *United States Statutes At Large* 112 (Oct. 1998), pp. 2860–2918. URL: https://www.govinfo.gov/content/pkg/STATUTE-112/pdf/STATUTE-112-Pg2860.pdf.

[2] Martín Abadi et al. "Control-flow integrity: Principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13 (Oct. 2009). DOI: 10.1145/1609956.1609960.

[3] Bert Abrath et al. "Tightly-coupled Self-debugging Software Protection". In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. SSPREW '16. Los Angeles, California, USA: ACM, 2016, 7:1–7:10. ISBN: 978-1-4503-4841-6. DOI: 10.1145/3015135.3015142. URL: http://doi.acm.org/10.1145/3015135.3015142.

[4] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. "A Taxonomy of Software Integrity Protection Techniques". In: *Advances in Computers* (2018).

[5] Rajendra Akerkar and Priti Sajja. "Components of KBS". In: *Knowledge-based systems*. Jones & Bartlett Publishers, 2010. Chap. 1.7, pp. 19–20.

[6] Debra Anderson, Thane Frivold, and Alfonso Valdes. *Next-generation Intrusion Detection Expert System (NIDES) – A Summary*. Tech. rep. Menlo Park, CA, USA: SRI International, June 1995.

[7] Brad Arkin, Scott Stender, and Gary McGraw. "Software penetration testing". In: *IEEE Security & Privacy* 3.1 (2005), pp. 84–87.

[8] Frederik Armknecht et al. "A security framework for the analysis and design of software attestation". In: Nov. 2013, pp. 1–12. DOI: 10.1145/2508859.2516650.

[9] David Aucsmith. "Tamper resistant software: An implementation". In: *International Workshop on Information Hiding*. Springer. 1996, pp. 317–333.

[10] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.

[11] Boaz Barak et al. "On the (Im)possibility of Obfuscating Programs". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–18. ISBN: 978-3-540-44647-7.

[12] Cataldo Basile, ed. *ASPIRE Framework Report*. Oct. 2016. URL: https://aspire-fp7.eu/sites/default/files/D5.11-ASPIRE-Framework-Report.pdf.

[13] Cataldo Basile, Daniele Canavese, and Leonardo Regano. "ADSS Validation". In: *ASPIRE Project Deliverable 1.06: ASPIRE Validation*. Jan. 2016, pp. 50–61. URL: https://aspire-fp7.eu/sites/default/files/D1.06-ASPIRE-Validation-v1.01.pdf.

[14] Cataldo Basile, Alberto Cappadonia, and Antonio Lioy. "Network-level access control policy analysis and transformation". In: *IEEE/ACM Transactions on Networking (TON)* 20.4 (2012), pp. 985–998.

[15] Cataldo Basile et al. “A meta-model for software protections and reverse engineering attacks”. In: *Journal of Systems and Software* 150 (2019), pp. 3–21.

[16] Cataldo Basile et al. “Automatic Discovery of Software Attacks via Backward Reasoning”. In: *Proceedings of the 1st International Workshop on Software Protection*. SPRO ’15. Florence, Italy: IEEE Press, 2015, pp. 52–58. URL: http://dl.acm.org/citation.cfm?id=2821429.2821443.

[17] Cataldo Basile et al. “On the impossibility of effectively using likely-invariants for software attestation purposes”. In: 2 (June 2018), pp. 1–25. DOI: 10.22667/JOWUA.2018.06.30.001.

[18] D. Bauer and M. Koblentz. “NIDX-an expert system for real-time network intrusion detection”. In: *1988 Computer Networking Symposium*. Vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 1988, pp. 98–106. DOI: 10.1109/CNS.1988.4983. URL: https://www.computer.org/csdl/proceedings-article/cns/1988/00004983/12OmNAkWvIU.

[19] Pietro Belotti et al. “Mixed-integer nonlinear optimization”. In: *Acta Numerica* 22 (2013), pp. 1–131. DOI: 10.1017/S0962492913000032.

[20] Tim Berners-Lee et al. “World Wide Web”. In: *Computers in Physics* 8.3 (1994), pp. 298–299.

[21] A. Bezzina et al. “A fair cluster-based resource and power allocation scheme for two-tier LTE femtocell networks”. In: *2016 Global Information Infrastructure and Networking Symposium (GIIS)*. Oct. 2016, pp. 1–6. DOI: 10.1109/GIIS.2016.7814945.

[22] David Binkley and Dawn Lawrie. “Development: Information Retrieval Applications”. In: *Encyclopedia of Software Engineering*. Ed. by Philip A. Laplante. CRC Press, 2010, pp. 231–307.

[23] Sue Black. “The Role of Ripple Effect in Software Evolution”. In: June 2006, pp. 249–268. ISBN: 9780470871829. DOI: 10.1002/0470871822.ch12.

[24] Emile Borel. “La théorie du jeu et les equation intégrales à noyau symétrique gauche.” comptes Rendus de l’Académie des sciences, 173: 1304–08. Translated by LJ Savage in”. In: *Econometrica* 21 (1921), pp. 97–100.

[25] Jerome Bracken and Martin Shubik. “Worldwide Nuclear Coalition Games: A Valuation of Strategic Offensive and Defensive Forces”. In: *Operations Research* 41.4 (1993), pp. 655–668. DOI: 10.1287/opre.41.4.655. URL: https://doi.org/10.1287/opre.41.4.655.

[26] Colin Bradley and Bernadette Currie. “Advances in the Field of Reverse Engineering”. In: *Computer-Aided Design and Applications* 2.5 (Aug. 2005), pp. 697–706. DOI: 10.1080/16864360.2005.10739029.

[27] Alessandro Cabutto et al. “Software Protection with Code Mobility”. In: *Proceedings of the Second ACM Workshop on Moving Target Defense*. MTD ’15. Denver, Colorado, USA: ACM, 2015, pp. 95–103. ISBN: 978-1-4503-3823-3. DOI: 10.1145/2808475.2808481. URL: http://doi.acm.org/10.1145/2808475.2808481.

[28] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu. “Deep Blue”. In: *Artif. Intell.* 134.1-2 (Jan. 2002), pp. 57–83. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. URL: http://dx.doi.org/10.1016/S0004-3702(01)00129-1.

[29] Daniele Canavese et al. “Estimating Software Obfuscation Potency with Artificial Neural Networks”. In: *International Workshop on Security and Trust Management*. Springer. 2017, pp. 193–202.

[30] Carlos Carrillo and Rafael Capilla. "Ripple Effect to Evaluate the Impact of Changes in Architectural Design Decisions". In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ECSA '18. Madrid, Spain: ACM, 2018, 41:1–41:8. ISBN: 978-1-4503-6483-6. DOI: 10.1145/3241403.3241446. URL: http://doi.acm.org/10.1145/3241403.3241446.

[31] M. Ceccato et al. "How Professional Hackers Understand Protected Code while Performing Attack Tasks". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. May 2017, pp. 154–164. DOI: 10.1109/ICPC.2017.2.

[32] Hoi Chang and Mikhail Atallah. "Protecting Software Code by Guards". In: Nov. 2001, pp. 160–175. DOI: 10.1007/3-540-47870-1_10.

[33] Stanley Chow et al. "White-box cryptography and an AES implementation". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2002, pp. 250–270.

[34] Shannon Claude. "Programming a Computer for Playing Chess". In: *Philosophical Magazine, Ser* 7.41 (1950), p. 314.

[35] Frederick B Cohen. "Operating system protection through program evolution". In: *Computers and Security* 12.6 (1993), pp. 565–584.

[36] George Coker et al. "Principles of remote attestation". In: *Int. J. Inf. Sec.* 10 (June 2011), pp. 63–81. DOI: 10.1007/s10207-011-0124-7.

[37] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Computer Science Technical Reports 148. Department of Computer Science, The University of Auckland, New Zealand, July 1997. URL: http://hdl.handle.net/2292/3491.

[38] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 184–196 (Nov. 1997). DOI: 10.1145/268946.268962.

[39] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[40] Council of the European Union. "Council Directive 91/250/EEC of 14 May 1991 on the legal protection of computer programs". In: *Official Journal* L 122 (May 1991), pp. 42–46. ISSN: 0378-6978. URL: https://publications.europa.eu/en/publication-detail/-/publication/92d68447-ea9a-4554-9540-de517984c310%22.

[41] Marie E. Csete and John C. Doyle. "Reverse Engineering of Biological Complexity". In: *Science* 295.5560 (2002), pp. 1664–1669. ISSN: 0036-8075. DOI: 10.1126/science.1069981.

[42] Christian Cudonnec, Philippe Jutel, and Paul Hariyanto. "Reaction". In: *ASPIRE Project Deliverable 3.06: Remote Attestation and Server Mobile Code Report*. June 2016, pp. 12–22. URL: https://aspire-fp7.eu/sites/default/files/D3.06-Remote-Attestation-and-Server-Mobile-Code-Report.pdf.

[43] Do Cuong et al. "Game Theory for Cyber Security and Privacy". In: *ACM Computing Surveys* 50 (May 2017), pp. 1–37. DOI: 10.1145/3057268.

[44] Michael A. Cusumano. "The Apple-Samsung Lawsuits". In: *Commun. ACM* 56.1 (Jan. 2013), pp. 28–31. ISSN: 0001-0782. DOI: 10.1145/2398356.2398366. URL: http://doi.acm.org/10.1145/2398356.2398366.

[45] Mila Dalla Preda et al. "Opaque Predicates Detection by Abstract Interpretation". In: *Algebraic Methodology and Software Technology*. Ed. by Michael Johnson and Varmo Vene. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 81–95. ISBN: 978-3-540-35636-3.

[46] Quynh H Dang. "Secure hash standard". In: *Federal Information Processing Standard* 180-4 (Aug. 2015). Ed. by National Institute of Standards and Technology.

[47] Tobias Dantzig. *Number: the Language of Science*. New York, NY, US, 1930.

[48] Bjorn De Sutter and Bart Coppens. "Anti-callback Stack Checks". In: *ASPIRE Project Deliverable 2.08: Offline Code Protection Report*. Nov. 2015, pp. 46–47. URL: https://aspire-fp7.eu/sites/default/files/D2.08-ASPIRE-Offline-Code-Protection-Report.pdf.

[49] Dorothy Denning and Peter G. Neumann. *Requirements and model for IDES – a real-time intrusion-detection expert system*. Tech. rep. Menlo Park, CA, USA: SRI International, Aug. 1985.

[50] Ozgur Depren et al. "An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks". In: *Expert Systems with Applications* 29.4 (2005), pp. 713–722. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2005.05.002. URL: http://www.sciencedirect.com/science/article/pii/S0957417405000989.

[51] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2011. ISBN: 978-1-118-07976-8. URL: https://www.wiley.com/en-us/Reversing%3A+Secrets+of+Reverse+Engineering+-p-9780764574818.

[52] Andreas Ekelhart, Stefan Fenz, and Thomas Neubauer. "Ontology-based decision support for information security risk management". In: *2009 Fourth International Conference on Systems*. IEEE. 2009, pp. 80–85.

[53] Pasi Eronen and Jukka Zitting. "An expert system for analyzing firewall rules". In: *Proceedings of the 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*. Nov. 2001, pp. 100–107.

[54] Hui Fang et al. "Multi-stage Binary Code Obfuscation Using Improved Virtual Machine". In: *Information Security*. Ed. by Xuejia Lai, Jianying Zhou, and Hui Li. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 168–181. ISBN: 978-3-642-24861-0.

[55] Edward A Feigenbaum and Bruce G Buchanan. "DENDRAL and Meta-DENDRAL: roots of knowledge systems and expert system applications". In: *Artificial Intelligence* 59.1-2 (1993), pp. 233–240.

[56] Stefan Fenz et al. "FORISK: Formalizing information security risk and compliance management". In: *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE. 2013, pp. 1–4.

[57] M. L. Fredman et al. "Storing a sparse table with O(1) worst case access time". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. Nov. 1982, pp. 165–169. DOI: 10.1109/SFCS.1982.39.

[58] Bill Gates et al. "An open letter to hobbyists". In: *Homebrew Computer Club Newsletter* 2.1 (1976), p. 2.

[59] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473.

[60] James R. Gosler. "Software Protection: Myth or Reality?" In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by Hugh C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 140–157. ISBN: 978-3-540-39799-1.

[61] Amar Gupta and H-MD Toong. "The first decade of personal computers". In: *Proceedings of the IEEE* 72.3 (1984), pp. 246–258.

[62] Maurice Howard Halstead. *Elements of Software Science*. Ed. by Elsevier. 1977. ISBN: 0-444-00205-7.

[63] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-10686-8.

[64] Lance J Hoffman. "Risk analysis and computer security: bridging the cultural gaps". In: *Proceedings of the 9th National Computer Security Conference*. 1986, pp. 156–161.

[65] Alfred Horn. "On sentences which are true of direct unions of algebras". In: *Journal of Symbolic Logic* 16.1 (1951), pp. 14–21. DOI: 10.2307/2268661.

[66] Shih-Kun Huang et al. "Software Crash Analysis for Automatic Exploit Generation on Binary Programs". In: *Reliability, IEEE Transactions on* 63 (Mar. 2014), pp. 270–289. DOI: 10.1109/TR.2014.2299198.

[67] *Information technology – Programming languages – Prolog – Part 1: General core*. Standard ISO/IEC 13211-1:1995. Geneva, CH: International Organization for Standardization and International Electrotechnical Commission, June 1995.

[68] Joint Task Force Transformation Initiative. *SP 800-39. Managing Information Security Risk: Organization, Mission, and Information System View*. Tech. rep. Gaithersburg, MD, United States, 2011.

[69] K.A. Jackson, D.H. Dubois, and C.A. Stallings. "An expert system application for network intrusion detection". In: (Jan. 1991).

[70] Peter Jackson. *Introduction to Expert Systems*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201876868.

[71] J. Jaffar and J.-L. Lassez. "Constraint Logic Programming". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: ACM, 1987, pp. 111–119. ISBN: 0-89791-215-2. DOI: 10.1145/41625.41635. URL: http://doi.acm.org/10.1145/41625.41635.

[72] M. Jang, H. Kim, and Y. Yun. "Detection of DLL Inserted by Windows Malicious Code". In: *2007 International Conference on Convergence Information Technology (ICCIT 2007)*. Nov. 2007, pp. 1059–1064. DOI: 10.1109/ICCIT.2007.320.

[73] Lalana Kagal et al. *A security architecture based on trust management for pervasive computing systems*. Tech. rep. MARYLAND UNIV BALTIMORE DEPT OF COMPUTER SCIENCE and ELECTRICAL ENGINEERING, 2005.

[74] Markus Kammerstetter, Christian Platzer, and Gilbert Wondracek. "Vanity, Cracks and Malware: Insights into the Anti-copy Protection Ecosystem". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 809–820. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382282. URL: http://doi.acm.org/10.1145/2382196.2382282.

[75] Arun Narayanan Kandanchatha and Yongxin Zhou. "System and method for obscuring bit-wise and two's complement integer computations in software". US Patent 7966499. July 2005. URL: https://patents.google.com/patent/US7966499.

[76] Joseph P Kearney et al. "Software complexity measurement". In: *Communications of the ACM* 29.11 (1986), pp. 1044–1050.

[77] Jung-Sun Kim, Minsoo Kim, and Bong-Nam Noh. "A Fuzzy Expert System for Network Forensics". In: *Computational Science and Its Applications – ICCSA 2004*. Ed. by Antonio Laganá et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 175–182. ISBN: 978-3-540-24707-4.

[78] Allan R Klumpp. "Apollo lunar descent guidance". In: *Automatica* 10.2 (1974), pp. 133–146.

[79] Donald E Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[80] Kyriakos Kritikos and Philippe Massonet. "An Integrated Meta-model for Cloud Application Security Modelling". In: *Procedia Computer Science* 97 (2016). 2nd International Conference on Cloud Forward: From Distributed to Complete Computing, pp. 84–93. ISSN: 1877-0509.

[81] S. Kumar et al. "Malware in Pirated Software: Case Study of Malware Encounters in Personal Computers". In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. Aug. 2016, pp. 423–427. DOI: 10.1109/ARES.2016.101.

[82] Tímea László and Ákos Kiss. "Obfuscating C++ Programs via Control Flow Flattening". In: *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica* 30 (June 2007).

[83] B. Li and Y. C. Kim. "Efficient routing and spectrum allocation considering QoT in elastic optical networks". In: *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*. July 2015, pp. 109–112. DOI: 10.1109/TSP.2015.7296233.

[84] Niandong Liao, Shengfeng Tian, and Tinghua Wang. "Network Forensics Based on Fuzzy Logic and Expert System". In: *Comput. Commun.* 32.17 (Nov. 2009), pp. 1881–1892. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2009.07.013. URL: http://dx.doi.org/10.1016/j.comcom.2009.07.013.

[85] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. "A Compiler-based Infrastructure for Software-protection". In: *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. PLAS '08. Tucson, AZ, USA: ACM, 2008, pp. 33–44. ISBN: 978-1-59593-936-4. DOI: 10.1145/1375696.1375702. URL: http://doi.acm.org/10.1145/1375696.1375702.

[86] Cullen Linn and Saumya Debray. "Obfuscation of executable code to improve resistance to static disassembly". In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pp. 290–299.

[87] T. F. Lunt et al. "IDES: a progress report (Intrusion-Detection Expert System)". In: *Proceedings of the Sixth Annual Computer Security Applications Conference*. Dec. 1990, pp. 273–285. DOI: 10.1109/CSAC.1990.143786.

[88] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[89] K. Metwally, A. Jarray, and A. Karmouch. "MILP-Based Approach for Efficient Cloud IaaS Resource Allocation". In: *2015 IEEE 8th International Conference on Cloud Computing*. June 2015, pp. 1058–1062. DOI: 10.1109/CLOUD.2015.152.

[90] Jelena Mirkovic and Peter Reiher. "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms". In: *SIGCOMM Comput. Commun. Rev.* 34.2 (Apr. 2004), pp. 39–53. ISSN: 0146-4833. DOI: 10.1145/997150.997156. URL: http://doi.acm.org/10.1145/997150.997156.

[91] Tom M Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, Mar. 1997. ISBN: 0-070-42807-7.

[92] T. Mouelhiv, F. Fleurey, and B. Baudry. "A Generic Metamodel For Security Policies Mutation". In: *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. Apr. 2008, pp. 278–286.

[93] Roger B. Myerson. *Game theory - Analysis of Conflict*. Harvard University Press, 1997. ISBN: 978-0-674-34116-6. URL: http://www.hup.harvard.edu/catalog/MYEGAM.html.

[94] Arthur N. Rasmussen, John F. Muratore, and Troy A. Heindel. "The INCO Expert System Project: CLIPS in Shuttle mission control". In: *First CLIPS Conference Proceedings*. Vol. 1. Feb. 1990, pp. 305–319.

[95] George C. Necula et al. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Compiler Construction*. Ed. by R. Nigel Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 213–228. ISBN: 978-3-540-45937-8.

[96] Michael L. Nelson. *A Survey of Reverse Engineering and Program Comprehension*. Tech. rep. cs.SE/0503068. European Organization for Nuclear Research (CERN), Mar. 2005. URL: http://cds.cern.ch/record/829443.

[97] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. "MulVAL: A Logic-based Network Security Analyzer." In: *USENIX security symposium*. Vol. 8. Baltimore, MD. 2005, pp. 113–128.

[98] Stephen F. Owens and Reuven R. Levary. "An Adaptive Expert System Approach for Intrusion Detection". In: *Int. J. Secur. Netw.* 1.3/4 (Dec. 2006), pp. 206–217. ISSN: 1747-8405. DOI: 10.1504/IJSN.2006.011780. URL: http://dx.doi.org/10.1504/IJSN.2006.011780.

[99] *OWL 2 Web Ontology Language New Features and Rationale (Second Edition)*. W3C Recommendation. Cambridge, MA, US: World Wide Web Consortium (W3C), Dec. 2012. URL: https://www.w3.org/TR/owl2-new-features/.

[100] Zhi Song Pan et al. "An integrated model of intrusion detection based on neural network and expert system". In: *17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05)*. Nov. 2005, pp. 672–673. DOI: 10.1109/ICTAI.2005.36.

[101] Karl Pearson. "The problem of the random walk". In: *Nature* 72.1867 (1905), p. 342.

[102] J Pournelle. *Zenith Z-100, Epson QX-10, software licensing, and the software piracy problem*. 1983.

[103] Todd A. Proebsting. "Optimizing an ANSI C Interpreter with Superoperators". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 322–332. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199526. URL: http://doi.acm.org/10.1145/199448.199526.

[104] Leonardo Regano et al. "Towards Automatic Risk Analysis and Mitigation of Software Applications". In: *IFIP International Conference on Information Security Theory and Practice*. Springer. 2016, pp. 120–135.

[105] Leonardo Regano et al. "Towards Optimally Hiding Protected Assets in Software Applications". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2017, pp. 374–385.

[106] Rolf Rolles. "Unpacking Virtualization Obfuscators". In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. WOOT'09. Montreal, Canada: USENIX Association, 2009, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1855876.1855877.

[107] Karu Sankaralingam et al. *Optimization and Mathematical Modeling in Computer Architecture*. Morgan & Claypool, 2013, pp. 144–. ISBN: 9781627052108. DOI: 10.2200/S00531ED1V01Y201308CAC026.

[108] Sebastian Schrittwieser et al. "Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis?" In: *ACM Comput. Surv.* 49.1 (Apr. 2016), 4:1–4:37.

[109] Ehab S Al-Shaer and Hazem H Hamed. "Management and translation of filtering security policies". In: *IEEE International Conference on Communications, 2003. ICC'03.* Vol. 1. IEEE. 2003, pp. 256–260.

[110] Carl Shapiro. "The theory of business strategy". In: *The Rand journal of economics* 20.1 (1989), pp. 125–137.

[111] Edward H. Shortliffe and Bruce G. Buchanan. "A model of inexact reasoning in medicine". In: *Mathematical Biosciences* 23.3 (1975), pp. 351–379. ISSN: 0025-5564. DOI: https://doi.org/10.1016/0025-5564(75)90047-4. URL: http://www.sciencedirect.com/science/article/pii/0025556475900474.

[112] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[113] S. E. Smaha. "Haystack: an intrusion detection system". In: *[Proceedings 1988] Fourth Aerospace Computer Security Applications*. Sept. 1988, pp. 37–44. DOI: 10.1109/ACSAC.1988.113412.

[114] T. Sommestad, M. Ekstedt, and H. Holm. "The Cyber Security Modeling Language: A Tool for Assessing the Vulnerability of Enterprise System Architectures". In: *IEEE Systems Journal* 7.3 (Sept. 2013), pp. 363–373. ISSN: 1932-8184.

[115] John F. Sowa. "Top-level ontological categories". In: *International Journal of Human-Computer Studies* 43.5 (1995), pp. 669–685. ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.1995.1068. URL: http://www.sciencedirect.com/science/article/pii/S1071581985710683.

[116] Lance Spitzner. *Honeypots: tracking hackers*. Vol. 1. Addison-Wesley Reading, 2003.

[117] Laura P Swiler and Cynthia Phillips. *A graph-based system for network-vulnerability analysis*. Tech. rep. Sandia National Labs., Albuquerque, NM (United States), 1998.

[118] Paolo Tonella et al. "POSTER: A Measurement Framework to Quantify Software Protections". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1505–1507. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2662360. URL: http://doi.acm.org/10.1145/2660267.2662360.

[119] Gene Tsudik and Rita C. Summers. "AudES - An Expert System for Security Auditing". In: *Proceedings of the The Second Conference on Innovative Applications of Artificial Intelligence*. IAAI '90. AAAI Press, 1991, pp. 221–232. ISBN: 0-262-68068-8. URL: http://dl.acm.org/citation.cfm?id=645450.653063.

[120] United States Department of Defense. *US Army Reverse Engineering Handbook (Guidelines and Procedures)*. 2016. URL: https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=53897.

[121] M. Välja et al. "Integrated Metamodel for Security Analysis". In: *48th Hawaii Int'l Conf. on System Sciences*. Jan. 2015, pp. 5192–5200.

[122] L. Van Put et al. "DIABLO: a reliable, retargetable and extensible link-time rewriting framework". In: *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.* Dec. 2005, pp. 7–12. DOI: 10.1109/ISSPIT.2005.1577061.

[123] Alessio Viticchié et al. "Assessment of source code obfuscation techniques". In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2016, pp. 11–20.

[124] Alessio Viticchié et al. “Reactive attestation: Automatic detection and reaction to software tampering attacks”. In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. ACM. 2016, pp. 73–84.

[125] Stijn Volckaert, Bjorn De Sutter, and Bert Abrath. “Self-debugging”. EU Patent 3330859. June 2018. URL: https://patents.google.com/patent/EP3330859.

[126] Chenxi Wang et al. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Tech. rep. Charlottesville, VA, USA, 2000.

[127] Richard Wartell et al. “Differentiating Code from Data in x86 Binaries”. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Dimitrios Gunopulos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 522–536. ISBN: 978-3-642-23808-6.

[128] Paul J Werbos. “Applications of advances in nonlinear sensitivity analysis”. In: *System modeling and optimization*. Springer, 1982, pp. 762–770.

[129] P. H. Winston and K. A. Prendergast. “XCON: An Expert Configuration System at Digital Equipment Corporation”. In: *The AI Business: Commercial Uses of Artificial Intelligence*. MITP, 1986. ISBN: 9780262257220. URL: https://ieeexplore.ieee.org/document/6284805.

[130] JM Wuerth. “The evolution of Minuteman guidance and control”. In: *Navigation* 23 (1976), pp. 64–75.

[131] Brecht Wyseur. “ASPIRE Protocol”. In: *ASPIRE Project Deliverable 1.04: Reference Architecture*. Aug. 2014, pp. 9–15. URL: https://aspire-fp7.eu/sites/default/files/D1.04-ASPIRE-Reference-Architecture-v2.1.pdf.

[132] Yongxin Zhou et al. “Information Hiding in Software with Mixed Boolean-Arithmetic Transforms”. In: *Information Security Applications*. Ed. by Sehun Kim, Moti Yung, and Hyung-Woo Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75. ISBN: 978-3-540-77535-5.